

Microsoft Visual C# Step by Step

Tenth Edition



 Professional

John Sharp

Microsoft Visual C# Step by Step

Tenth Edition

John Sharp

MICROSOFT VISUAL C# STEP BY STEP, TENTH EDITION

Published with the authorization of Microsoft Corporation by:
Pearson Education, Inc.

Copyright © 2022 by Agylia Group Ltd.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-13-761983-2

ISBN-10: 0-13-761983-9

Library of Congress Control Number: 2022930224

ScoutAutomatedPrintCode

TRADEMARKS

Microsoft and the trademarks listed at <http://www.microsoft.com> on the "Trademarks" webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

WARNING AND DISCLAIMER

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author, the publisher, and Microsoft Corporation shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the programs accompanying it.

SPECIAL SALES

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corp-sales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Editor-in-Chief: Brett Bartow

Executive Editor: Loretta Yates

Sponsoring Editor: Charvi Arora

Development Editor: Kate Shoup

Managing Editor: Sandra Schroeder

Senior Project Editor: Tracey Croom

Production Editor: Dan Foster

Copy Editor: Dan Foster

Indexer: Valerie Haynes Perry

Proofreader: Kim Wimpsett

Technical Editor: David Franson

Editorial Assistant: Cindy Teeters

Cover Designer: Twist Creative, Seattle

Compositor: Danielle Foster

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

Please contact us with concerns about any potential bias at <https://www.pearson.com/report-bias.html>.

Contents at a Glance

<i>Acknowledgments</i>	<i>xxi</i>
<i>About the author</i>	<i>xxiii</i>
<i>Introduction</i>	<i>xxv</i>

PART I INTRODUCING MICROSOFT VISUAL C# AND MICROSOFT VISUAL STUDIO 2022

CHAPTER 1	Welcome to C#	3
CHAPTER 2	Working with variables, operators, and expressions	45
CHAPTER 3	Writing methods and applying scope	73
CHAPTER 4	Using decision statements	111
CHAPTER 5	Using compound assignment and iteration statements	133
CHAPTER 6	Managing errors and exceptions	153

PART II UNDERSTANDING THE C# OBJECT MODEL

CHAPTER 7	Creating and managing classes and objects	181
CHAPTER 8	Understanding values and references	205
CHAPTER 9	Creating value types with enumerations and structures	231
CHAPTER 10	Using arrays	251
CHAPTER 11	Understanding parameter arrays	277
CHAPTER 12	Working with inheritance	289
CHAPTER 13	Creating interfaces and defining abstract classes	311
CHAPTER 14	Using garbage collection and resource management	339

PART III UNDERSTANDING THE C# OBJECT MODEL

CHAPTER 15	Implementing properties to access fields	365
CHAPTER 16	Handling binary data and using indexers	395
CHAPTER 17	Introducing generics	413
CHAPTER 18	Using collections	445
CHAPTER 19	Enumerating collections	469
CHAPTER 20	Decoupling application logic and handling events	483

CHAPTER 21	Querying in-memory data by using query expressions	513
CHAPTER 22	Operator overloading	537
PART IV	BUILDING UNIVERSAL WINDOWS PLATFORM APPLICATIONS WITH C#	
CHAPTER 23	Improving throughput by using tasks	559
CHAPTER 24	Improving response time by performing asynchronous operations	599
CHAPTER 25	Implementing the user interface for a Universal Windows Platform app	641
CHAPTER 26	Displaying and searching for data in a Universal Windows Platform app	687
CHAPTER 27	Accessing a remote database from a Universal Windows Platform app	717
	<i>Index</i>	771

Contents

<i>Acknowledgments</i>	<i>xxi</i>
<i>About the author</i>	<i>xxiii</i>
<i>Introduction</i>	<i>xxv</i>

PART I INTRODUCING MICROSOFT VISUAL C# AND MICROSOFT VISUAL STUDIO 2022

Chapter 1	Welcome to C#	3
	Writing your first C# program	3
	Beginning programming with the Visual Studio 2022 environment	9
	Writing your first program using Visual Studio 2022	14
	Using namespaces	21
	Namespaces and assemblies	23
	Commenting code	24
	Creating a graphical application	24
	Examining the Universal Windows Platform app	37
	Adding code to the graphical application	40
	Summary	43
	Quick Reference	44
Chapter 2	Working with variables, operators, and expressions	45
	Understanding statements	45
	Using identifiers	46
	Identifying keywords	46
	Using variables	48
	Naming variables	48
	Declaring variables	48
	Specifying numeric values	49

Using optional parameters and named arguments	100
Defining optional parameters	102
Passing named arguments	102
Resolving ambiguities with optional parameters and named arguments	103
Summary	108
Quick reference	109
 Chapter 4 Using decision statements	 111
Declaring Boolean variables	111
Using Boolean operators	112
Understanding equality and relational operators	112
Understanding conditional logical operators	113
Short-circuiting	114
Summarizing operator precedence and associativity	114
Pattern matching	115
Using if statements to make decisions	116
Understanding if statement syntax	116
Using blocks to group statements	117
Cascading if statements	118
Using switch statements	124
Understanding switch statement syntax	124
Following the switch statement rules	125
Using switch expressions with pattern matching	129
Summary	131
Quick reference	132
 Chapter 5 Using compound assignment and iteration statements	 133
Using compound assignment operators	133
Writing while statements	135
Writing for statements	140

Writing do statements	142
Summary	151
Quick reference	151

Chapter 6 Managing errors and exceptions 153

Trying code and catching exceptions	154
Unhandled exceptions	155
Using multiple catch handlers	156
Catching multiple exceptions.....	157
Filtering exceptions.....	158
Propagating exceptions.....	163
Using checked and unchecked integer arithmetic	165
Writing checked statements.....	166
Writing checked expressions	166
Throwing exceptions	170
Using throw expressions	174
Using a finally block	175
Summary	176
Quick reference	177

PART II UNDERSTANDING THE C# OBJECT MODEL

Chapter 7 Creating and managing classes and objects 181

Understanding classification	181
The purpose of encapsulation.....	182
Defining and using a class.....	182
Controlling accessibility.....	184
Working with constructors.....	186
Overloading constructors.....	187
Deconstructing an object	196

Chapter 9	Creating value types with enumerations and structures	231
	Working with enumerations.....	231
	Declaring an enumeration	231
	Using an enumeration	232
	Choosing enumeration literal values.....	233
	Choosing an enumeration's underlying type	233
	Working with structures	236
	Declaring a structure	237
	Understanding differences between structures and classes	238
	Declaring structure variables	240
	Understanding structure initialization	240
	Copying structure variables	245
	Summary	248
	Quick reference	249
 Chapter 10	 Using arrays	 251
	Declaring array variables.....	252
	Creating an array instance	252
	Populating and using an array.....	253
	Creating an implicitly typed array	254
	Accessing an individual array element	255
	Accessing a series of array elements	255
	Iterating through an array.....	256
	Passing arrays as parameters or return values for a method	257
	Copying arrays	259
	Using multidimensional arrays	260
	Creating jagged arrays.....	261
	Accessing arrays that contain value types	272
	Summary	275
	Quick reference	275

Chapter 11	Understanding parameter arrays	277
	Overloading: a recap	277
	Using array arguments	278
	Declaring a params array	279
	Using params object[]	281
	Using a params array	283
	Comparing parameter arrays and optional parameters	286
	Summary	288
	Quick reference	288
 Chapter 12	 Working with inheritance	 289
	What is inheritance?	289
	Using inheritance	290
	The System.Object class revisited	292
	Calling base-class constructors	292
	Assigning classes	293
	Declaring new methods	295
	Declaring virtual methods	296
	Declaring override methods	297
	Understanding protected access	300
	Creating extension methods	305
	Summary	309
	Quick reference	310
 Chapter 13	 Creating interfaces and defining abstract classes	 311
	Understanding interfaces	311
	Defining an interface	312
	Implementing an interface	313
	Referencing a class through its interface	314
	Working with multiple interfaces	315
	Explicitly implementing an interface	316
	Handling versioning with interfaces	318
	Interface restrictions	319
	Defining and using interfaces	320

Abstract classes	329
Abstract methods	330
Sealed classes	330
Sealed methods	331
Implementing and using an abstract class	331
Summary	337
Quick reference	338

Chapter 14 Using garbage collection and resource management 339

The life and times of an object	339
Writing finalizers	340
Why use the garbage collector?	343
How does the garbage collector work?	344
Recommendations	345
Resource management	345
Disposal methods	346
Exception-safe disposal	346
The using statement and the IDisposable interface	347
Calling the Dispose method from a finalizer	349
Implementing exception-safe disposal	351
Handling asynchronous disposal	359
Summary	361
Quick reference	361

PART III UNDERSTANDING THE C# OBJECT MODEL

Chapter 15 Implementing properties to access fields 365

Implementing encapsulation by using methods	366
What are properties?	367
Using properties	370
Read-only properties	371
Write-only properties	371
Property accessibility	372

Understanding property restrictions.	373
Declaring interface properties.	375
Replacing methods with properties.	376
Pattern-matching with properties	380
Generating automatic properties.	381
Initializing objects by using properties.	383
Automatic properties and immutability.	385
Using records with properties to implement lightweight structures	388
Summary	392
Quick reference	393
 Chapter 16 Handling binary data and using indexers	 395
What is an indexer?	395
Storing binary values	396
Displaying binary values	397
Manipulating binary values	397
Solving the same problems using indexers	398
Understanding indexer accessors.	400
Comparing indexers and arrays	401
Indexers in interfaces	403
Using indexers in a Windows application.	404
Summary	410
Quick reference	410
 Chapter 17 Introducing generics	 413
The problem: Issues with the object type	413
The generics solution	417
Generics vs. generalized classes	419
Generics and constraints	419
Creating a generic class	420
The theory of binary trees.	420
Building a binary tree class by using generics.	423

Creating a generic method.....	433
Defining a generic method to build a binary tree.....	434
Variance and generic interfaces	438
Covariant interfaces	440
Contravariant interfaces	441
Summary	444
Quick reference	444

Chapter 18 Using collections 445

What are collection classes?.....	445
The List<T> collection class.....	447
The LinkedList<T> collection class	449
The Queue<T> collection class	450
The PriorityQueue<TElement, TPriority> collection class.....	451
The Stack<T> collection class.....	452
The Dictionary<TKey, TValue> collection class.....	453
The SortedList<TKey, TValue> collection class.....	454
The HashSet<T> collection class	455
Using collection initializers.....	457
Find methods, predicates, and lambda expressions.....	458
The forms of lambda expressions.....	460
Lambda expressions and anonymous methods.....	461
Comparing arrays and collections	462
Summary	466
Quick reference	467

Chapter 19 Enumerating collections 469

Enumerating the elements in a collection	469
Manually implementing an enumerator.....	470
Implementing the IEnumerable interface	475
Implementing an enumerator by using an iterator.....	477
A simple iterator.....	478
Defining an enumerator for the Tree<TItem> class by using an iterator.....	479

Summary	482
Quick reference	482
Chapter 20 Decoupling application logic and handling events	483
Understanding delegates	484
Examples of delegates in the .NET class library	485
The automated factory scenario.....	487
Declaring and using delegates	490
Lambda expressions and delegates.....	499
Enabling notifications by using events	500
Declaring an event.....	501
Subscribing to an event	502
Unsubscribing from an event.....	502
Raising an event	502
Understanding user-interface events	503
Using events	504
Summary	510
Quick reference	511
Chapter 21 Querying in-memory data by using query expressions	513
What is LINQ?	513
Using LINQ in a C# application	514
Selecting data	516
Filtering data	518
Ordering, grouping, and aggregating data.....	519
Joining data.....	521
Using query operators	522
Querying data in Tree<TItem> objects.....	525
LINQ and deferred evaluation.....	530
Summary	534
Quick reference	534

Chapter 22	Operator overloading	537
	Understanding operators	537
	Operator constraints	538
	Overloaded operators	539
	Creating symmetric operators	540
	Understanding compound assignment evaluation	541
	Declaring increment and decrement operators	542
	Comparing operators in structures and classes	542
	Defining operator pairs	543
	Implementing operators	544
	Overriding the equality operators	547
	Understanding conversion operators	550
	Providing built-in conversions	551
	Implementing user-defined conversion operators	552
	Creating symmetric operators, revisited	553
	Writing conversion operators	553
	Summary	555
	Quick reference	556

PART IV BUILDING UNIVERSAL WINDOWS PLATFORM APPLICATIONS WITH C#

Chapter 23	Improving throughput by using tasks	559
	Why perform multitasking by using parallel processing?	559
	The rise of the multicore processor	560
	Implementing multitasking by using Microsoft .NET	561
	Tasks, threads, and the ThreadPool	562
	Creating, running, and controlling tasks	563
	Using the Task class to implement parallelism	566
	Abstracting tasks by using the Parallel class	576
	When not to use the Parallel class	580

Canceling tasks and handling exceptions	582
The mechanics of cooperative cancellation.....	582
Handling task exceptions by using the AggregateException class.....	594
Using continuations with canceled and faulted tasks	596
Summary	596
Quick reference	597

Chapter 24 Improving response time by performing asynchronous operations 599

Implementing asynchronous methods.....	600
Defining asynchronous methods: the problem	600
Defining asynchronous methods: the solution.....	603
Defining asynchronous methods that return values.....	609
Asynchronous method pitfalls.....	610
Asynchronous methods and the Windows Runtime APIs.....	611
Tasks, memory allocation, and efficiency.....	613
Using PLINQ to parallelize declarative data access.....	616
Using PLINQ to improve performance while iterating through a collection	616
Canceling a PLINQ query	621
Synchronizing concurrent access to data.....	621
Locking data	625
Synchronization primitives for coordinating tasks	625
Canceling synchronization	627
The concurrent collection classes.....	628
Using a concurrent collection and a lock to implement thread-safe data access	629
Summary	639
Quick reference	639

Chapter 25 Implementing the user interface for a Universal Windows Platform app 641

Features of a Universal Windows Platform app	643
Using the Blank App template to build a Universal Windows Platform app	645

Implementing a scalable user interface	649
Implementing a tabular layout by using a Grid control	659
Adapting the layout by using the Visual State Manager	667
Applying styles to a UI	674
Summary	685
Quick reference	686
Chapter 26 Displaying and searching for data in a Universal Windows Platform app	687
Implementing the Model-View-ViewModel pattern	687
Displaying data by using data binding	688
Modifying data by using data binding	694
Using data binding with a ComboBox control	699
Creating a ViewModel	702
Adding commands to a ViewModel	706
Summary	715
Quick reference	716
Chapter 27 Accessing a remote database from a Universal Windows Platform app	717
Retrieving data from a database	717
Creating an entity model	728
Creating and using a REST web service	735
Updating the UWP application to use the web service	752
Searching for data in the Customers app	761
Inserting, updating, and deleting data through a REST web service	767
Summary	769
Quick reference	770
Index	771

Acknowledgments

Ho boy! Welcome to the 10th edition. In the acknowledgments to previous editions, I have made references to painting the Forth Railway Bridge and Sisyphus pushing the rock as never-ending tasks. In the future, maybe the role of updating *Microsoft C# Step By Step* will be added to this legendary list. That said, writing and updating books is far more rewarding than wielding a brush or rolling a stone up a hill forever and a day, with the added bonus that I can retire at some point.

Despite the fact that my name is on the cover, authoring a book such as this is far from a one-man project. I'd like to thank the following people who have provided unstinting support and assistance throughout this endeavor.

First, Loretta Yates at Pearson Education, who took on the role of prodding me into action and ever-so-gently tying me down to well-defined deliverables and hand-off dates. Without her initial impetus and cajoling, this project would not have gotten off the ground.

Next, Charvi Arora and her tireless team of editors, especially Kate Shoup and Dan Foster, who ensured that my grammar remained at least semi-acceptable and picked up on the missing words and nonsense phrases in the text. Also, David Fransen, who had the unenviable task of reviewing and testing the code and exercises. I know from experience that this can be a thankless and frustrating task, but the hours spent and the resulting feedback can only make for a better book. Of course, any errors that remain are entirely my responsibility, and I am happy to listen to feedback from any reader.

As ever, I must also thank Diana, my better half, who keeps me sane, fed, and watered. During Covid-19 lockdown, she felt that our house wasn't crowded enough, so she brought two rather manic kittens into the family. The dogs are now terrified, but we have endless hours of fun putting the curtains back up and playing "hunt the mouse/frog/spider or whatever they have captured and brought indoors." I wouldn't have home-life any other way.

And lastly, to James and Frankie, who have both now flown the nest. James has spent the last couple of years working for the British government in Manila (he says). Judging by the photos, it seems more like he has been on a touring holiday of the beaches of Southeast Asia. Frankie has remained closer to home so she can pop in and catch the mice/frogs/spiders from time to time. By the way, to those developers she manages at her place of work, it's time for you to make her a cup of tea!

About the author

JOHN SHARP is a principal technologist for CM Group Ltd, part of the Civica Group, a software development and consultancy company in the United Kingdom. He is well versed as a software consultant, developer, author, and trainer, with more than 35 years of experience, ranging from Pascal programming on CP/M and C/Oracle application development on various flavors of UNIX to the design of C# and JavaScript distributed applications and development on Windows 11 and Microsoft Azure. He also spends much of his time writing courseware for Microsoft, focusing on areas such as data science using R and Python, big data processing with Spark and CosmosDB, SQL Server, NoSQL, web services, Blazor, cross-platform development with frameworks such as Xamarin and MAUI, and scalable application architectures with Azure.

Introduction

A lot has changed in the last 20 years. For a laugh, I sometimes retrieve my copy of *Microsoft C# Step By Step*, first edition, released in 2001, and wonder at my naive innocence back in those days. Surely, C# was the peak of programming language perfection at that time. C# and the .NET Framework hit the world of development with a bang, and the reverberations continue to this day. However, rather than dying away, they rumble through software development with increased significance. Rather than being a single-platform approach as the naysayers of 2001 originally screamed, C# and .NET have shown themselves to be a complete multiplatform solution, whether you're building applications for Windows, macOS, Linux, or Android. Additionally, C# and .NET have proved themselves the runtime of choice for many cloud-based systems. Where would Azure be without them?

In the past, most common programming languages went through occasional updates, often spread several years apart. For example, if you look at Fortran, you will see standards named Fortran 66, Fortran 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008, and Fortran 2018. That's seven updates in the last 55 years. While this relatively slow cycle of change promotes stability, it can also lead to stagnation. The issue is that the nature of problems that developers must address changes rapidly, and the tools they depend on should ideally keep pace so that they can develop effective solutions. Microsoft .NET provides a continually evolving framework, and C# undergoes frequent updates to make the best use of the platform. So, in contrast to Fortran, C# has undergone a rapid evolution since it was first released—six versions in the last five years alone, with another update due in 2022. The C# language still supports code written 20+ years ago, but these days the additions and enhancements to the language enable you to create solutions using more elegant code and concise constructs. For this reason, I make periodic updates to this book; this is now the 10th edition!

If you're interested, the following list contains a brief history of C#:

- C# 1.0 made its public debut in 2001.
- C# 2.0, with Visual Studio 2005, provided several important new features, including generics, iterators, and anonymous methods.
- C# 3.0, which was released with Visual Studio 2008, added extension methods, lambda expressions, and, most famously of all, the Language-Integrated Query (LINQ) facility.

- C# 4.0, released in 2010, provided further enhancements that improved its interoperability with other languages and technologies. These features included support for named and optional arguments and the dynamic type, which indicates that the language runtime should implement late binding for an object. Important additions to the .NET Framework, released concurrently with C# 4.0, were the classes and types that constitute the Task Parallel Library (TPL). Using the TPL, you can build highly scalable applications that can take full advantage of multicore processors.
- C# 5.0 added native support for asynchronous task-based processing through the `async` method modifier and the `await` operator.
- C# 6.0 was an incremental upgrade with features designed to make life simpler for developers. These features included items such as string interpolation (you need never use `String.Format` again!), enhancements to the ways in which properties are implemented, expression-bodied methods, and others.
- C# 7.0 through 7.3 added further enhancements to aid productivity and remove some of the minor anachronisms of C#. For example, these versions enabled you to implement property accessors as expression-bodied members, methods can return multiple values in the form of tuples, the use of `out` parameters was simplified, and `switch` statements were extended to support pattern- and type-matching. These versions of the language also included many other smaller tweaks to address concerns that many developers had, such as allowing the `Main` method to be asynchronous.
- C# 8.0, C# 9.0, and C# 10.0 continue this theme of enhancing the language to improve readability and aid developer productivity. Some major additions included records, which you can use to build immutable reference types; extensions to pattern matching, enabling you to use this feature throughout the language and not just in `switch` statements; top-level statements, which enable you to use C# as a scripting language (you don't always need to write a `Main` method); default interface methods; static local functions; asynchronous disposable types; and many other features, which are covered in this book.

It goes without saying that Microsoft Windows is an important platform for running C# applications, but now you can also run code developed by using C# on other operating systems, such as Linux, through the .NET runtime. This opens up possibilities for writing code that can run in multiple environments. Additionally, Windows supports highly interactive applications that can share data and collaborate as well as connect to services running in the cloud. The key notion in Windows is Universal Windows Platform (UWP) apps—applications designed to run on any Windows 10 or Windows 11 device, whether a full-fledged desktop system, a laptop, a tablet, or even an Internet of Things (IoT) device with limited resources. Once you've mastered the core features of C#, gaining the skills to build applications that can run on all these platforms is critical.

The cloud has become such an important element in the architecture of many systems—ranging from large-scale enterprise applications to mobile apps running on portable devices—that I decided to focus on this aspect of development in the final chapter of the book.

The development environment provided by Visual Studio makes these features easy to use, and the many new wizards and enhancements included in the latest version of Visual Studio can greatly improve your productivity as a developer. I hope you have as much fun working through this book as I had writing it!

Who should read this book

This book assumes that you are a developer who wants to learn the fundamentals of programming with C# by using Visual Studio and the .NET version 6 or later. By the time you complete this book, you will have a thorough understanding of C# and will have used it to build responsive and scalable applications that can run on the Windows operating system.

Who should not read this book

This book is aimed at developers new to C# but not completely new to programming. As such, it concentrates primarily on the C# language. This book is not intended to provide detailed coverage of the multitude of technologies available for building enterprise-level and global applications for Windows, such as ADO.NET, ASP.NET, Azure, or Windows Communication Foundation. If you require more information on any of these items, you might consider reading some of the other titles available from Microsoft Press.

Finding your best starting point in this book

This book is designed to help you build skills in several essential areas. You can use this book if you're new to programming or if you're switching from another programming language such as C, C++, Java, or Visual Basic. Use the following table to find your best starting point.

If you are	Follow these steps
New to object-oriented programming	1. Install the practice files as described in the upcoming section, "Code samples." 2. Work through Chapters 1 to 22 sequentially. 3. Complete Chapters 23 to 27 as your level of experience and interest dictates.
Familiar with procedural programming languages, such as C, but new to C#	1. Install the practice files as described in the upcoming section, "Code samples." 2. Skim the first five chapters to get an overview of C# and Visual Studio 2022, and then concentrate on Chapters 6 through 22. 3. Complete Chapters 23 to 27 as your level of experience and interest dictates.
Migrating from an object-oriented language such as C++ or Java	1. Install the practice files as described in the upcoming section, "Code samples." 2. Skim the first seven chapters to get an overview of C# and Visual Studio 2022, and then concentrate on Chapters 8 through 22. 3. For information about building Universal Windows Platform applications, read Chapters 23 to 27.
Switching from Visual Basic to C#	1. Install the practice files as described in the upcoming section, "Code samples." 2. Work through Chapters 1 to 22 sequentially. 3. For information about building Universal Windows Platform applications, read Chapters 23 to 27. 4. Read the "Quick reference" sections at the end of the chapters for information about specific C# and Visual Studio 2022 constructs.
Referencing the book after working through the exercises	1. Use the index or the table of contents to find information about particular subjects. 2. Read the "Quick reference" sections at the end of each chapter to find a brief review of the syntax and techniques presented in the chapter.

Most of the book's chapters include hands-on samples that let you try out the concepts you just learned. No matter which sections you choose to focus on, be sure to download and install the sample applications on your system.

Conventions and features in this book

This book presents information by using conventions designed to make the information readable and easy to follow.

- Each exercise consists of a series of tasks, presented as numbered steps (1, 2, and so on) listing each action you must take to complete the exercise.
- Boxed elements with labels such as "Note," "Tip," "Important," and "More Info" provide additional information or alternative methods for completing a step successfully.

- Text that you type (apart from code blocks) and screen elements you select appear in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, “Press Alt+Tab” means that you hold down the Alt key while you press the Tab key.

System requirements

You will need the following hardware and software to complete the practice exercises in this book:

- Windows 10 (Home, Professional, Education, or Enterprise) or Windows 11 (Home, Professional, Education, or Enterprise).
- The most recent build of Visual Studio Community 2022, Visual Studio Professional 2022, or Visual Studio Enterprise 2022. (Make sure that you have installed any updates.) As a minimum, you should select the following workloads when installing Visual Studio 2022:
 - Universal Windows Platform development
 - .NET desktop development
 - ASP.NET and web development
 - Azure development
 - Data storage and processing
 - .NET Core cross-platform development



Note All the exercises and code samples in this book have been developed and tested using Visual Studio Community 2022. They should all work, unchanged, in Visual Studio Professional 2022 and Visual Studio Enterprise 2022.

- 1.8 GHz or faster 64-bit processor; quad-core or better recommended. ARM processors are not supported.
- 4 GB of RAM.
- Hard disk space: minimum of 850 MB up to 210 GB of available space, depending on features installed; typical installations require 20 to 50 GB of free space.

- Video card that supports a minimum display resolution of 720p (1280 by 720); Visual Studio will work best at a resolution of WXGA (1366 by 768) or higher.
- Internet connection to download software or chapter examples.

Depending on your Windows configuration, you might require local administrator rights to install or configure Visual Studio.

You also need to enable developer mode on your computer to be able to create and run UWP apps. For details on how to do this, see “Enable Your Device for Development,” at <https://msdn.microsoft.com/library/windows/apps/dn706236.aspx>.

Code samples

Most of the chapters in this book include exercises with which you can interactively try out new material learned in the main text. You can download all the sample projects, in both their pre-exercise and post-exercise formats, from the following page:

MicrosoftPressStore.com/VisualCsharp10e/downloads

Installing the code samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book:

1. Unzip the CSharpSBS.zip file that you downloaded from the book’s website, extracting the files into your Documents folder.
2. If prompted, review the end-user license agreement. If you accept the terms, select the Accept option and then click Next.

Note If the license agreement doesn’t appear, you can access it from the same webpage where you downloaded the CSharpSBS.zip file.

Using the code samples

Each chapter in this book explains when and how to use the code samples for that chapter. When it’s time to use a code sample, the book will list the instructions for how to open the files.

Important Many of the code samples depend on NuGet packages that are not included with the code. These packages are downloaded automatically the first time you build a project. As a result, if you open a project and examine the code before doing a build, Visual Studio might report a large number of errors for unresolved references. Building the project will resolve these references, and the errors should disappear.

If you'd like to know all the details, here's a list of the sample Visual Studio projects and solutions, grouped by the folders in which you can find them. In many cases, the exercises provide starter files and completed versions of the same projects that you can use as a reference. The completed projects for each chapter are stored in folders with the suffix "- Complete."

Project/Solution	Description
Chapter 1	
HelloWorld	This project gets you started. It steps through the creation of a simple program using a text editor. The program displays a text-based greeting.
HelloWorld2	This project demonstrates how to use the .NET Command Level Interface (CLI) to build and run a simple C# application.
TestHello	This is a Visual Studio project that displays a greeting.
HelloUWP	This project opens a window that prompts the user for his or her name and then displays a greeting.
Chapter 2	
PrimitiveDataTypes	This project demonstrates how to declare variables by using each of the primitive types, how to assign values to these variables, and how to display their values in a window.
MathOperators	This program introduces the arithmetic operators (+ - * / %).
Chapter 3	
Methods	In this project, you'll reexamine the code in the MathOperators project and investigate how it uses methods to structure the code.
DailyRate	This project walks you through writing your own methods, running the methods, and stepping through the method calls by using the Visual Studio 2015 debugger.
DailyRate Using Optional Parameters	This project shows you how to define a method that takes optional parameters and call the method by using named arguments.
Factorial	This project demonstrates a recursive method that calculates the factorial of a number.

Project/Solution	Description
Chapter 4	
Selection	This project shows you how to use a cascading if statement to implement complex logic, such as comparing the equivalence of two dates.
SwitchStatement	This simple program uses a switch statement to convert characters into their XML representations.
SwitchStatement using Pattern Matching	This is an amended version of the SwitchStatement project that uses pattern matching to simplify the logic in the switch statement.
Chapter 5	
WhileStatement	This project demonstrates a while statement that reads the contents of a source file one line at a time and displays each line in a text box on a form.
DoStatement	This project uses a do statement to convert a decimal number to its octal representation.
Chapter 6	
MathOperators	This project revisits the MathOperators project from Chapter 2 and shows how various unhandled exceptions can make the program fail. The try and catch keywords then make the application more robust so that it no longer fails.
Chapter 7	
Classes	This project covers the basics of defining your own classes, complete with public constructors, methods, and private fields. It also shows how to create class instances by using the new keyword and how to define static methods and fields.
Chapter 8	
Parameters	This program investigates the difference between value parameters and reference parameters. It demonstrates how to use the ref and out keywords.
Chapter 9	
StructsAndEnums	This project defines a struct type to represent a calendar date.
Chapter 10	
Cards	This project shows how to use arrays to model hands of cards in a card game.
Chapter 11	
ParamsArray	This project demonstrates how to use the params keyword to create a single method that can accept any number of int arguments.
Chapter 12	
Vehicles	This project creates a simple hierarchy of vehicle classes by using inheritance. It also demonstrates how to define a virtual method.
ExtensionMethod	This project shows how to create an extension method for the int type, providing a method that converts an integer value from base 10 to a different number base.

Project/Solution	Description
Chapter 13	
Drawing	This project implements part of a graphical drawing package. The project uses interfaces to define the methods that drawing shapes expose and implement.
Chapter 14	
GarbageCollectionDemo	This project shows how to implement exception-safe disposal of resources by using the Dispose pattern.
Chapter 15	
Drawing Using Properties	This project extends the application in the Drawing project developed in Chapter 13 to encapsulate data in a class by using properties.
AutomaticProperties	This project shows how to create automatic properties for a class and use them to initialize instances of the class.
Student enrollment	This project demonstrates how to use records to model structured immutable types.
Chapter 16	
Indexers	This project uses two indexers: one to look up a person's phone number when given a name and the other to look up a person's name when given a phone number.
Chapter 17	
BinaryTree	This solution shows you how to use generics to build a type-safe structure that can contain elements of any type.
BuildTree	This project demonstrates how to use generics to implement a type-safe method that can take parameters of any type.
Chapter 18	
Cards	This project updates the code from Chapter 10 to show how to use collections to model hands of cards in a card game.
Chapter 19	
BinaryTree	This project shows you how to implement the generic <code>IEnumerator<T></code> interface to create an enumerator for the generic <code>Tree</code> class.
IteratorBinaryTree	This solution uses an iterator to generate an enumerator for the generic <code>Tree</code> class.
Chapter 20	
Delegates	This project shows how to decouple a method from the application logic that invokes it by using a delegate. The project is then extended to show how to use an event to alert an object to a significant occurrence, and how to catch an event and perform any processing required.
Chapter 21	
QueryBinaryTree	This project shows how to use LINQ queries to retrieve data from a binary tree object.

Project/Solution	Description
Chapter 22	
ComplexNumbers	This project defines a new type that models complex numbers and implements common operators for this type.
Chapter 23	
GraphDemo	This project generates and displays a complex graph on a UWP form. It uses a single thread to perform the calculations.
Parallel GraphDemo	This version of the GraphDemo project uses the Parallel class to abstract out the process of creating and managing tasks.
GraphDemo with Cancellation	This project shows how to implement cancellation to halt tasks in a controlled manner before they have completed.
ParallelLoop	This application provides an example showing when you should not use the Parallel class to create and run tasks.
Chapter 24	
GraphDemo	This is a version of the GraphDemo project from Chapter 23 that uses the async keyword and the await operator to perform the calculations that generate the graph data asynchronously.
PLINQ	This project shows some examples of using PLINQ to query data by using parallel tasks.
CalculatePi	This project uses a statistical sampling algorithm to calculate an approximation for pi. It uses parallel tasks.
ParallelTest	This program illustrates the dangers of allowing uncontrolled data access to shared data by parallel threads.
Chapter 25	
Customers	This project implements a scalable user interface that can adapt to different device layouts and form factors. The user interface applies XAML styling to change the fonts and background image displayed by the application.
Chapter 26	
DataBinding	This is a version of the Customers project that uses data binding to display customer information retrieved from a data source in the user interface. It also shows how to implement the INotifyPropertyChanged interface so that the user interface can update customer information and send these changes back to the data source.
ViewModel	This version of the Customers project separates the user interface from the logic that accesses the data source by implementing the Model-View-ViewModel pattern.

Project/Solution	Description
Chapter 27	
Web Service	This solution includes a web application that provides a REST web service that the Customers application uses to retrieve customer information and modify data held in a SQL Server database. The web service uses the Entity Framework to access the database. The database and the web service run using Azure.
Customers with insert and update features	This solution contains an updated version of the Customers project that uses the REST web service to create new customers and modify the details of existing customers.

Errata and book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at:

MicrosoftPressStore.com/VisualCsharp10e/errata

If you discover an error that is not already listed, please submit it to us at the same page.

For additional book support and information, please visit:

MicrosoftPressStore.com/Support

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to:

<http://support.microsoft.com>

Stay in touch

Let's keep the conversation going! We're on Twitter: <http://twitter.com/MicrosoftPress>.

PART I

Introducing Microsoft Visual C# and Microsoft Visual Studio 2022

CHAPTER 1	Welcome to C#	3
CHAPTER 2	Working with variables, operators, and expressions.....	45
CHAPTER 3	Writing methods and applying scope	73
CHAPTER 4	Using decision statements.....	111
CHAPTER 5	Using compound assignment and iteration statements.....	133
CHAPTER 6	Managing errors and exceptions.....	153

This introductory part of the book covers the essentials of the C# language and shows you how to get started building applications with Visual Studio 2022.

In Part I, you'll learn how to create new projects in Visual Studio and how to declare variables, use operators to create values, call methods, and write many of the statements you need when implementing C# programs. You'll also learn how to handle exceptions and how to use the Visual Studio debugger to step through your code and spot problems that prevent your applications from working correctly.

Welcome to C#

After completing this chapter, you will be able to:

- Create a C# console application.
- Use the Microsoft Visual Studio 2022 programming environment.
- Explain the purpose of namespaces.
- Create a simple graphical C# application.

This chapter introduces Visual Studio 2022, the programming environment and toolset designed to help you build applications for Microsoft Windows. Visual Studio 2022 is the ideal tool for writing C# code, and it provides many features that you'll learn about as you progress through this book. In this chapter, you'll use Visual Studio 2022 to build some simple C# applications and get started on the path to building highly functional solutions for Windows.

Writing your first C# program

The simplest way to start learning almost any programming language is to write the ubiquitous "Hello World!" application. This is a simple program that runs from the computer console and displays the message "Hello World!"



Note A console application is an application that runs in a command prompt window instead of providing a graphical user interface (GUI).

Visual Studio 2022 provides a graphical interactive development environment (IDE). The IDE provides editors, syntax checking, file management, and project organization tools among other items. It enables you to be highly productive but can be a little overwhelming at first. To keep matters simple initially, in the first couple of exercises in this chapter, you'll work from the Windows command prompt and use a set of tools called the .NET command-line interface (CLI). These tools are installed as part of Visual Studio 2022. You'll write C# programs using Notepad. Once you've created a couple of simple applications and have a basic understanding of how things work, you'll switch to the Visual Studio IDE.

To build and run the Hello World! application

1. On the Windows taskbar, select **Start**, type **Command**, and press **Enter**.
2. In the command prompt window, type **cd C:\Users\YourName\Documents\Microsoft Press\VCSBS\Chapter 1** to move to the folder containing the code for Chapter 1. Replace the text *YourName* in this path with your Windows username.



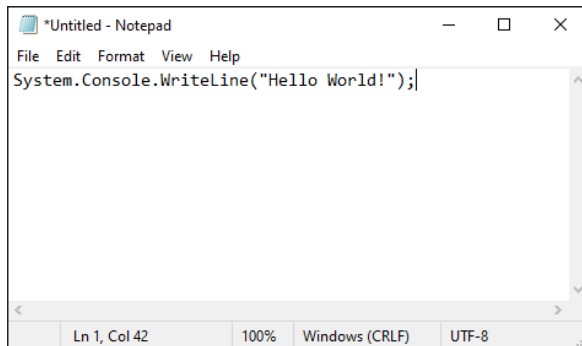
Note To avoid repetition and save space, throughout the rest of this book I'll refer to the path C:\Users\YourName\Documents simply as your *Documents* folder.

3. Run the following command to create a new folder named HelloWorld:

```
mkdir HelloWorld
```

4. Type **Notepad**, and press **Enter**.
5. In Notepad, type the following text:

```
System.Console.WriteLine("Hello World!");
```



In this line of code, the `WriteLine` function writes the string specified as the argument (the value between the parentheses) to a specific destination. The `WriteLine` function belongs to a class named `Console`, which corresponds to the screen. Any text passed to the `WriteLine` method of the `Console` class will be printed on the screen. The `Console` class belongs to a library of objects provided with C#. The items in this library are gathered into a namespace named `System`. The `System` namespace, and the corresponding library, contains several utility classes for performing fundamental operations in C#. You'll explore many of these classes as you proceed through this book.



Note Strictly speaking, functions that belong to classes, like `WriteLine`, should be called *methods*. You'll learn about methods starting in Chapter 3, "Writing methods and applying scope."

6. Open the **File** menu, select **Save As**, and save the file with the name **Program.cs** in the **Microsoft Press\VCSBS\Chapter 1\HelloWorld** folder in your **Documents** folder.



Note All C# code files should have the suffix `.cs`.

7. Open the **File** menu and select **New** to create a new, blank file.
8. Add the following text to this file:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>
</Project>
```

This is an example of a C# project file. The .NET CLI tools will use this file to compile your code into an executable application. The contents of this file specify the type of executable file to create (an EXE file), and the version of the .NET runtime to use to build and run the application (.NET 6.0). You'll see later in this book that you can create other types of executable files, such as dynamic-link libraries (DLLs), which can be shared by many applications.

9. Open the **File** menu, select **Save As**, and save the file with the name **Hello.csproj** in the **Microsoft Press\VCSBS\Chapter 1\HelloWorld** folder in your **Documents** folder.



Note C# project files should have the suffix `.csproj`.

10. Close Notepad, return to the command prompt window, and run the following command to move to the HelloWorld folder:
- ```
cd HelloWorld
```
11. Run the `dir` command and verify that the folder contains the two files `HelloWorld.csproj` and `Program.cs`.

```
Administrator: Command Prompt

C:\Users\student\Documents\Microsoft Press\VCSBS\Chapter 1\HelloWorld>dir
Volume in drive C is Windows
Volume Serial Number is DABB-41A5

Directory of C:\Users\student\Documents\Microsoft Press\VCSBS\Chapter 1\HelloWorld

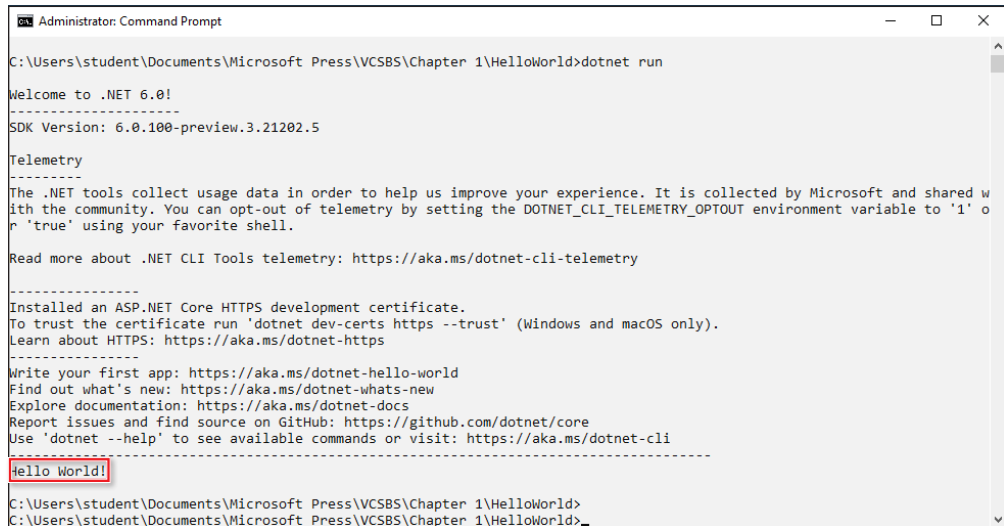
05/05/2021 04:05 PM <DIR> .
05/05/2021 04:05 PM <DIR> ..
05/05/2021 04:02 PM 167 Hello.csproj
05/05/2021 03:52 PM 41 Program.cs
 2 File(s) 208 bytes
 2 Dir(s) 82,347,581,440 bytes free

C:\Users\student\Documents\Microsoft Press\VCSBS\Chapter 1\HelloWorld>
```

12. Type the following command to build and run the application:

**dotnet run**

You'll see several messages while the .NET CLI tools download and install one or two additional libraries and a development certificate. At the end, you should see the text `Hello World!` This is the output from your program.



```
Administrator: Command Prompt

C:\Users\student\Documents\Microsoft Press\VCSBS\Chapter 1\HelloWorld>dotnet run

Welcome to .NET 6.0!

SDK Version: 6.0.100-preview.3.21202.5

Telemetry

The .NET tools collect usage data in order to help us improve your experience. It is collected by Microsoft and shared with the community. You can opt-out of telemetry by setting the DOTNET_CLI_TELEMETRY_OPTOUT environment variable to '1' or 'true' using your favorite shell.

Read more about .NET CLI Tools telemetry: https://aka.ms/dotnet-cli-telemetry

Installed an ASP.NET Core HTTPS development certificate.
To trust the certificate run 'dotnet dev-certs https --trust' (Windows and macOS only).
Learn about HTTPS: https://aka.ms/dotnet-https

Write your first app: https://aka.ms/dotnet-hello-world
Find out what's new: https://aka.ms/dotnet-whats-new
Explore documentation: https://aka.ms/dotnet-docs
Report issues and find source on GitHub: https://github.com/dotnet/core
Use 'dotnet --help' to see available commands or visit: https://aka.ms/dotnet-cli

Hello World!

C:\Users\student\Documents\Microsoft Press\VCSBS\Chapter 1\HelloWorld>
```

13. Repeat the `dotnet run` command. This time, the .NET CLI tools don't need to download any libraries or certificates because they're already installed. Instead, you should see only the message `Hello World!`

The Hello World! application is quick and easy to write, but you had to add the project file to enable the .NET CLI tools to understand how to build and run the program. This approach is fine for simple applications, but more complicated systems can require significant configuration and correspondingly complex project files. For example, an application might require several different libraries, and the project file must specify the names of these libraries and where the .NET CLI tools should download them from. Fortunately, the .NET CLI can automate much of this complexity, or at least make it easier to manage.

In the next exercise, you'll create another version of Hello World!, but this time you'll get the .NET CLI tools to generate the project file for you.

## To use the .NET CLI tools to build and run a C# project

1. In the command prompt window, return to the **Microsoft Press\VC SBS\Chapter 1\HelloWorld** folder in your **Documents** folder.

```
cd C:\Users\YourName\Documents\Microsoft Press\VC SBS\Chapter 1
```

2. Create a new folder named **HelloWorld2**.

```
mkdir HelloWorld2
```

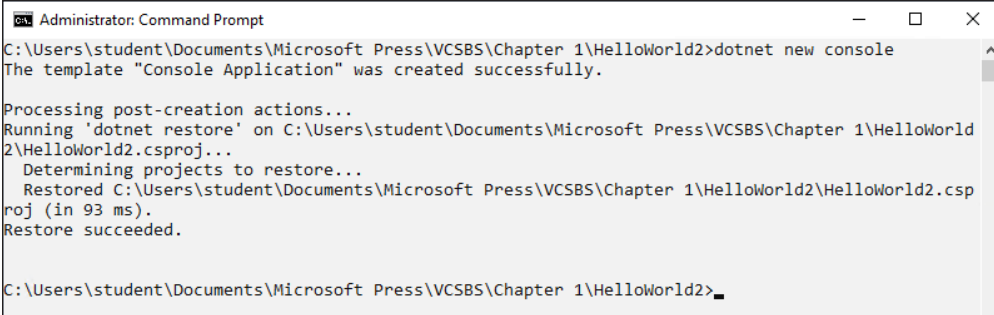
3. Move to the **HelloWorld2** folder.

```
cd HelloWorld2
```

4. Run the following command to create a new console application. The word `console` is the name of a template that the .NET CLI uses for generating the application.

```
dotnet new console
```

You'll see a few messages as the .NET CLI creates a C# project file for a new console application.



```
Administrator: Command Prompt
C:\Users\student\Documents\Microsoft Press\VC SBS\Chapter 1\HelloWorld2>dotnet new console
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\Users\student\Documents\Microsoft Press\VC SBS\Chapter 1\HelloWorld2\HelloWorld2.csproj...
 Determining projects to restore...
 Restored C:\Users\student\Documents\Microsoft Press\VC SBS\Chapter 1\HelloWorld2\HelloWorld2.csproj (in 93 ms).
Restore succeeded.

C:\Users\student\Documents\Microsoft Press\VC SBS\Chapter 1\HelloWorld2>
```

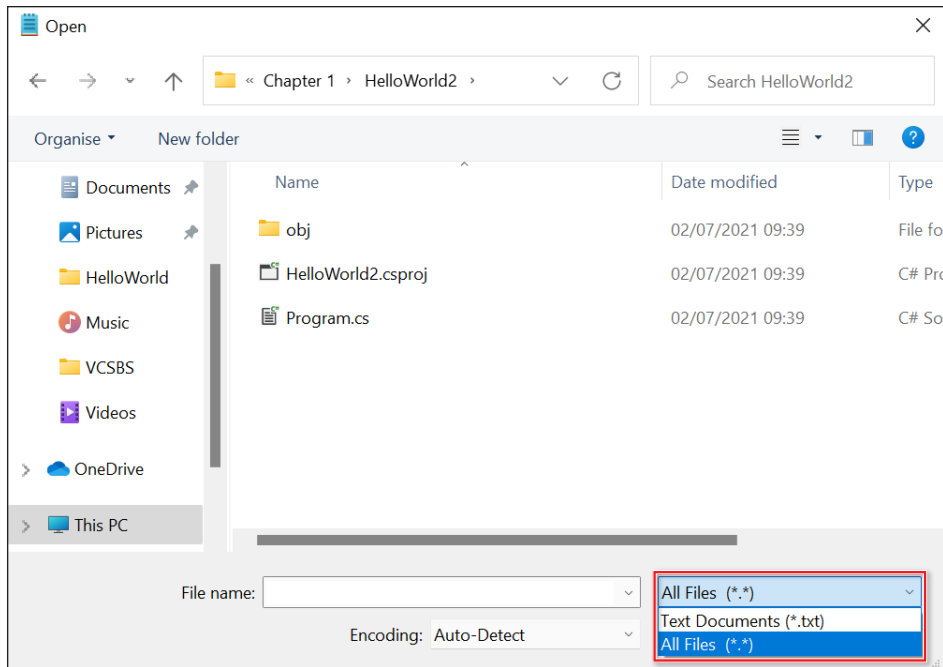


**Note** The .NET CLI provides several other templates for creating other types of applications, such as an ASP.NET web application or a Windows Forms graphical application.

5. Type **dir** to display the contents of the HelloWorld2 folder.

You'll see a project file named *HelloWorld2.csproj*, a C# code file named *Program.cs*, and a folder named *obj*. The *obj* folder contains configuration files and other bits of data that the .NET CLI will use to compile and run the application.

6. Start Notepad. Then open the **File** menu, select **Open**, and open the **HelloWorld2.csproj** file in the **HelloWorld2** folder. You may need to set the file type dropdown list box to **All Files** to see the files in the HelloWorld2 folder.



7. The contents of the HelloWorld2.csproj file should look like this:

```
<Project Sdk="Microsoft.NET.Sdk">
 <PropertyGroup>
 <OutputType>Exe</OutputType>
 <TargetFramework>net6.0</TargetFramework>
 </PropertyGroup>
</Project>
```

This project file is the same as the one that you created manually in the previous exercise.

8. Open the Program.cs file. It should contain the following code:

```
using System;

namespace HelloWorld2
{
 class Program
 {
 static void Main(string[] args)
 {
 Console.WriteLine("Hello World!");
 }
 }
}
```

This program file is a little more complicated than the one you created earlier, although it does exactly the same thing. The `Program.cs` file defines a class called `Program` that contains a method called `Main`. In C#, all executable code must be defined within a method, and all methods must belong to a class or a struct. You'll learn more about classes in Chapter 7, "Creating and managing classes and objects," and about structs in Chapter 9, "Creating value types with enumerations and structures."

The `Main` method designates the program's entry point. This method should be defined in the manner specified in the `Program` class as a static method; otherwise, the .NET Framework might not recognize it as the starting point for your application when you run it. (You'll look at methods in detail in Chapter 3, "Writing methods and applying scope." Chapter 7 provides more information on static methods.)

At this point, you're probably thinking, "I didn't create a `Program` class or a `Main` method" in the previous exercise. In the latest version of C#, if you have a simple application such as that shown in the first exercise, the C# compiler will create its own entry point if you don't provide one. (You can think of this as a compiler-synthesized `Main` method, although its name is actually `Main`.) This can save you writing unnecessary *boilerplate* code in many cases; just compare the preceding code to the program you wrote in the previous exercise to see what I mean. However, the console application template provided with the .NET CLI always creates a `Program` class and a `Main` method. Additionally, other types of applications use a different way of marking the entry point for a program that doesn't utilize a `Main` method. The templates for these types of applications generate the appropriate startup code. You'll see an example later in this chapter in the section "Creating a graphical application."

As for the `using` statement and the namespace definition, you'll learn more about these items later in this chapter.

9. Don't change anything; close Notepad, and return to the command prompt.
10. Compile and run the application. This is the same command you used in the previous exercise.

```
dotnet run
```

The result should be the message `Hello World!` displayed on the screen.

## Beginning programming with the Visual Studio 2022 environment

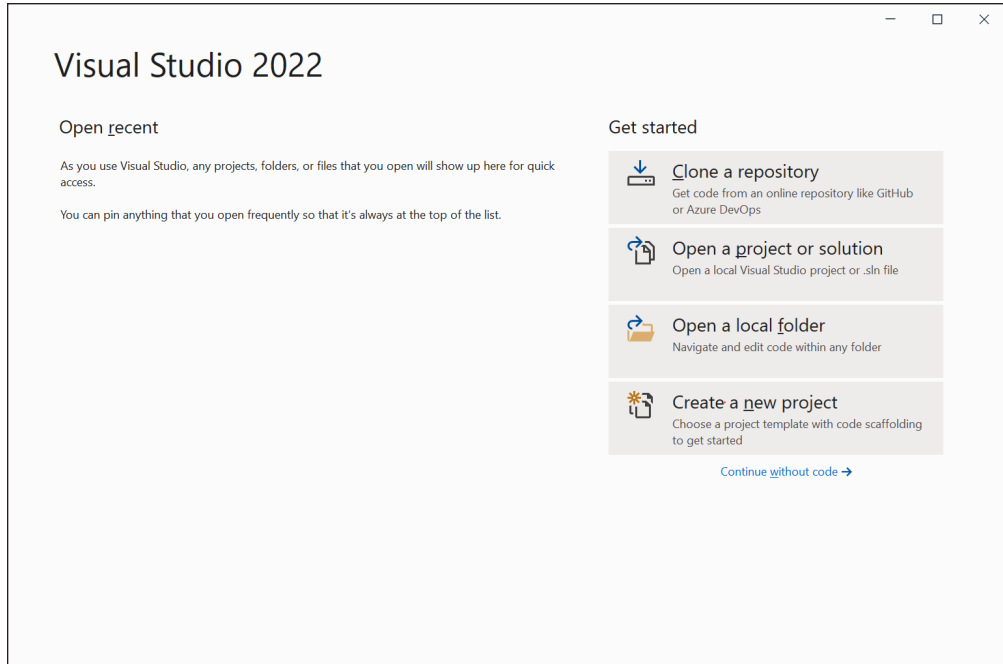
---

Now that you've seen the basic structure of a C# application, it's time to turn your attention to Visual Studio 2022. Visual Studio is a tool-rich development environment containing the functionality that you need to create large or small C# projects running on Windows. You can even construct projects that seamlessly combine modules written in different programming languages, such as C++, Visual Basic, and F#. In the next exercise, you'll open the Visual Studio 2022 development environment and create another version of the Hello World! console application.

## To create a console application in Visual Studio 2022

1. On the Windows taskbar, select **Start**, type **Visual Studio 2022**, and press **Enter**. Alternatively, select the **Visual Studio 2022** icon on the **Start** menu.

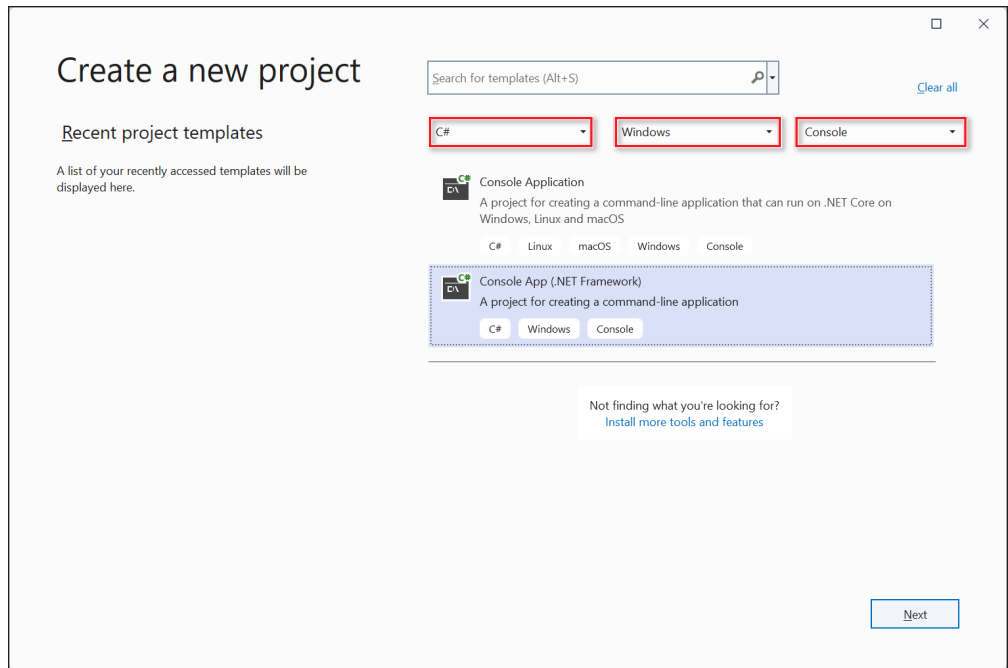
Visual Studio 2022 starts and displays the Start page, similar to the following image:



2. In the Get started pane, select **Create a new project**.

The Create a new project dialog opens. This dialog lists the templates that you can use as a starting point for building an application. The dialog categorizes templates according to the programming language you're using and the type of application.

3. In the Create a new project dialog, enter the following values, and select **Next**:
  - a. In the **All languages** dropdown list box, select **C#**.
  - b. In the **All platforms** dropdown list box, select **Windows**.
  - c. In the **All project types** dropdown list box, select **Console**.
  - d. Select the **Console App (.NET Framework)** template.



**Note** The .NET Framework version of the template uses an implementation of .NET that is specific to Windows. The .NET Core version of the Console Application template uses a version of .NET that is available not just on Windows but also on Linux and macOS. However, the .NET Framework version of the template is sufficient for this exercise.

4. In the Configure your new project dialog, enter the following values, and select **Create**:
  - a. In the **Project name** field, type **TestHello**.
  - b. In the **Location** field, type **C:\Users\YourName\Documents\Microsoft Press\VCSBS\Chapter 1\**.
  - c. Leave the **Solution name** box set to **TestHello**.
  - d. Ensure that the **Place solution and project in the same directory** checkbox is left unchecked.
  - e. Leave the **Framework** option set to its default value.



## Configure your new project

Console App (.NET Framework) C# Windows Console

Project name

Location

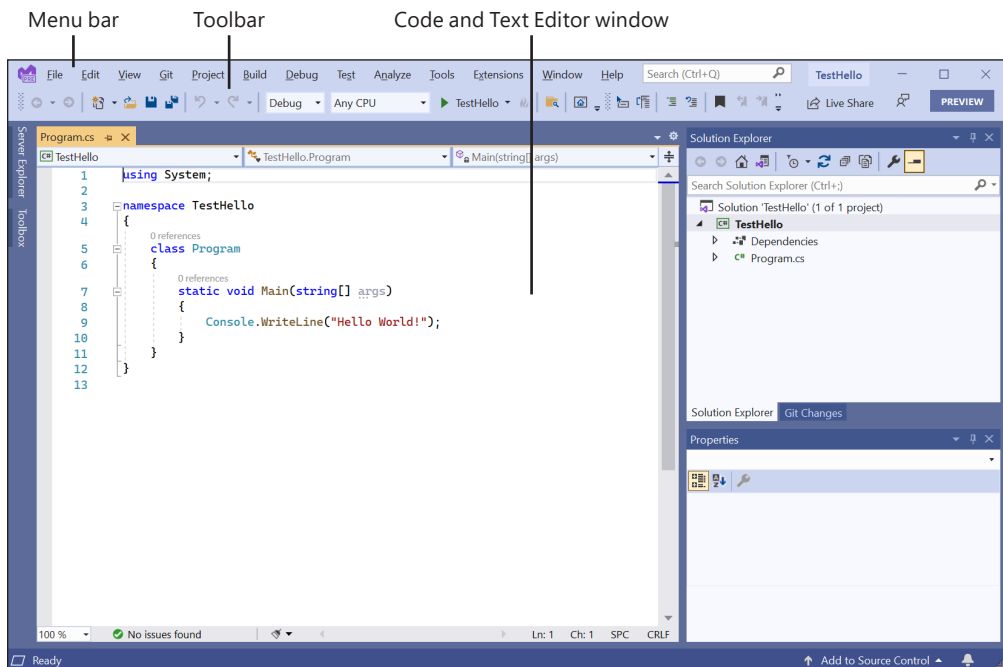
Solution

Solution name

☐ Place solution and project in the same directory

Framework

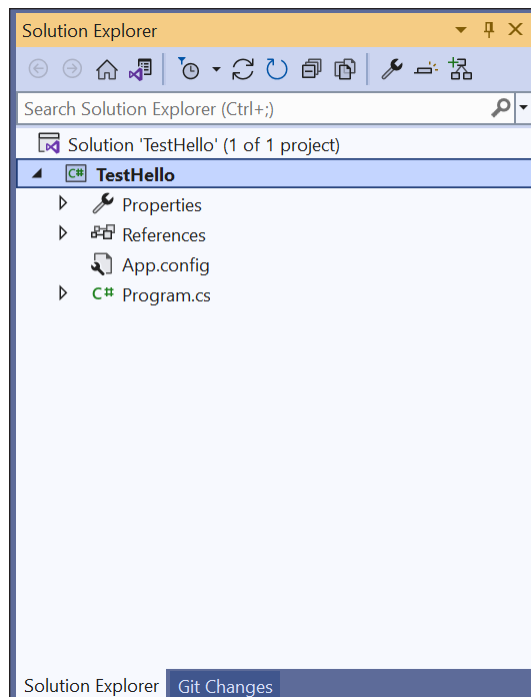
Visual Studio creates the project by using the Console Application template. Visual Studio then displays the starter code for the project, as in the following image.



The menu bar at the top of the screen provides access to the features you'll use in the programming environment. You can use the keyboard or mouse to access the menus and commands, exactly as you can in all Windows-based programs. The toolbar is located beneath the menu bar. It provides button shortcuts to run the most frequently used commands.

The Code and Text Editor window, occupying the main part of the screen, displays the contents of source files. In a multifile project, when you edit more than one file, each source file has its own tab labeled with the name of the source file. You can select the tab to bring the named source file to the foreground in the Code and Text Editor window.

The Solution Explorer pane appears on the right side of the IDE, adjacent to the Code and Text Editor window.



Solution Explorer displays the names of the files associated with the project, among other items. You can also double-click a file name in Solution Explorer to bring that source file to the foreground in the Code and Text Editor window.

5. Examine the files listed in Solution Explorer, which Visual Studio 2022 has created as part of your project:
  - **Solution 'TestHello'** This is the top-level solution file. Each application contains a single solution file. A solution can contain one or more projects, and Visual Studio 2022 creates the solution file to help organize these projects. If you use File Explorer to look at your Documents\Microsoft Press\VCSBS\Chapter 1\TestHello folder, you'll see that the actual name of this file is TestHello.sln.

- **TestHello** This is the C# project file. Each project file references one or more files containing the source code and other artifacts for the project, such as graphics images. You must write all the source code in a single project in the same programming language. In File Explorer, this file is actually called TestHello.csproj, and it's stored in the \Microsoft Press\VCSBS\Chapter 1\TestHello\TestHello folder in your Documents folder.
- **Properties** This is a folder in the TestHello project. If you expand it (select the arrow next to Properties), you'll see that it contains a file called AssemblyInfo.cs. This is a special file that you can use to add attributes to a program, such as the name of the author, the date the program was written, and so on. You can specify additional attributes to modify how the program runs. Explaining how to use these attributes is beyond the scope of this book.
- **References** This folder contains references to libraries of compiled code that your application can use. When your C# code is compiled, it is converted into a library and given a unique name. In the Microsoft .NET Framework, these libraries are called *assemblies*. Developers use assemblies to package useful functionality that they have written so that they can distribute it to other developers who might want to use these features in their own applications. If you expand the References folder, you'll see the default set of references that Visual Studio 2022 adds to your project. These assemblies provide access to many of the commonly used features of the .NET Framework and are provided by Microsoft with Visual Studio 2022. You'll learn about many of these assemblies as you progress through the exercises in this book.
- **App.config** This is the application configuration file. It is optional, and it might not always be present. You can specify settings that your application uses at runtime to modify its behavior, such as the version of the .NET Framework to use to run the application. You'll learn more about this file in later chapters of this book.
- **Program.cs** This is the C# source file generated by the template, and it's displayed in the Code and Text Editor window when the project is first created. You replace the contents of this file with your own application code.

## Writing your first program using Visual Studio 2022

---

You learned earlier that the Program.cs file defines a class called Program that contains a method called Main.



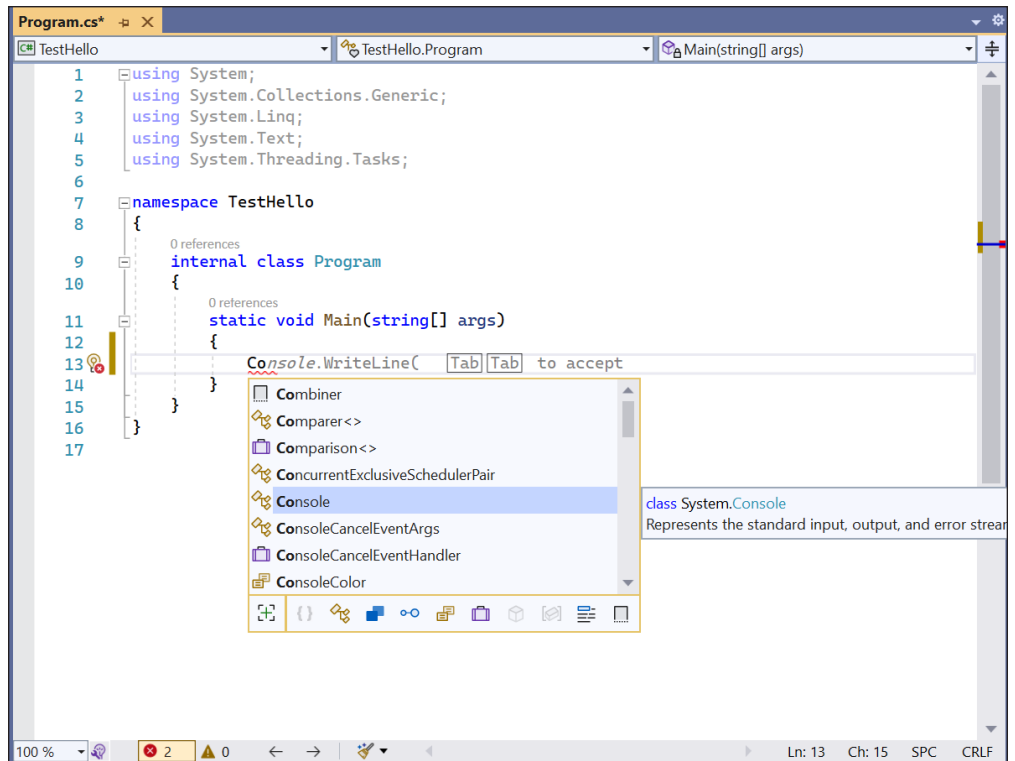
**Note** C# is a case-sensitive language. You must spell Main with an uppercase M.

In the following exercises, you'll modify the code to also display the message Hello YourName! to the console window. You'll build and run the application using Visual Studio. You'll also learn how namespaces are used to partition code elements.

## To write code in Visual Studio by using Microsoft IntelliSense

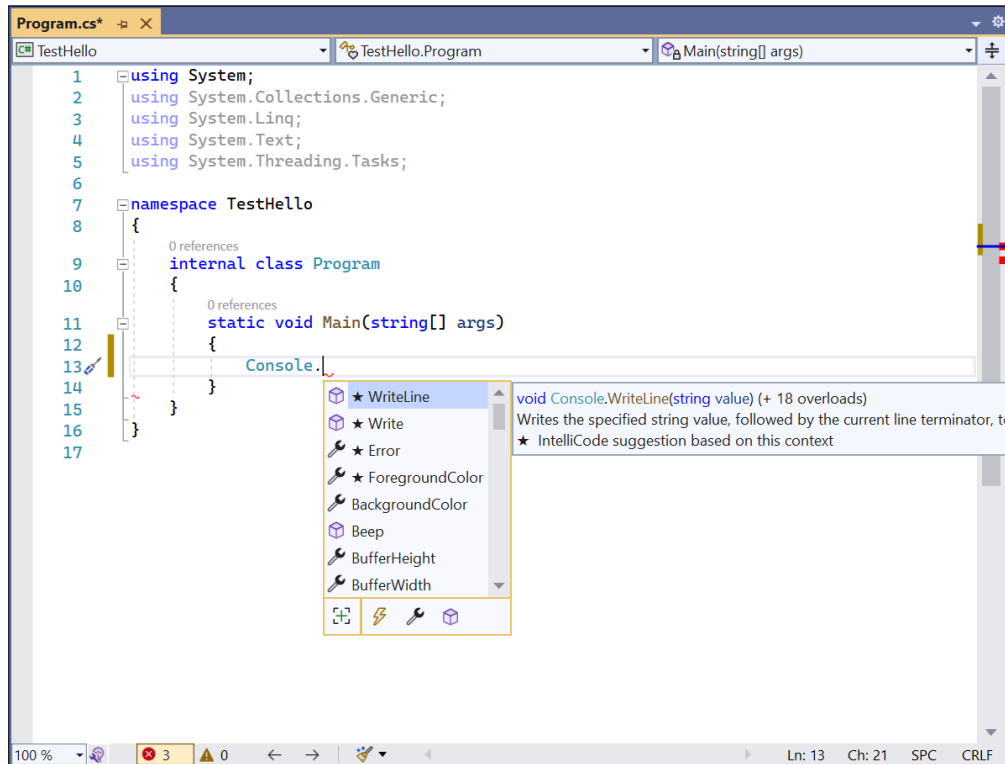
1. In the Code and Text Editor window displaying the Program.cs file, place the cursor in the Main method, immediately after the curly brace (}) on the line below the `static void Main(string[] args);` statement. Press **Enter** to create a new line.
2. On the new line, type the word **Console**. The `Console` class is provided by the assemblies referenced by your application. This class contains several methods for displaying messages in the console window and reading input from the keyboard, not just the `WriteLine` method.

As you type the letter C at the start of the word `Console`, an IntelliSense list appears. This list contains all the C# keywords and data types that are valid in this context. You can either continue typing or scroll through the list and double-click the `Console` item with the mouse. Alternatively, after you've typed **Cons**, the IntelliSense list automatically homes in on the `Console` item, and you can press the Tab or Enter key to select it.



3. Type a period immediately following `Console`.

Another IntelliSense list appears, displaying the methods, properties, and fields of the `Console` class.



4. Scroll down through the list, select **WriteLine**, and press **Enter**. Alternatively, press the **Tab** key to select the default method. (The default method is that one that's most commonly used.)

The IntelliSense list closes, and the word `WriteLine` is added to the source file. The new statement should look like this:

```
Console.WriteLine
```

5. Type `(` (if an open parenthesis hasn't already been added. Another IntelliSense tip will appear.

This tip displays the parameters that the `WriteLine` method can take. `WriteLine` is an overloaded method, meaning the `Console` class contains more than one method named `WriteLine`. In fact, it provides 18 different versions of this method. You can use each version of the `WriteLine` method to output different types of data. (Chapter 3 describes overloaded methods in more detail.) The new statement should now look like this:

```
Console.WriteLine(
```



**Tip** You can select the up and down arrows in the tip to scroll through the different overloads of `WriteLine`.

6. Type `);` and the statement should now look like this:

```
Console.WriteLine();
```

7. Move the cursor and type the string **"Hello YourName"** (including the quotation marks) between the left and right parentheses following the `WriteLine` method.

The statement now look like this:

```
Console.WriteLine("Hello YourName");
```



**Tip** Get into the habit of typing matched character pairs, such as parentheses ( and ) and curly braces { and } before filling in their contents. It's easy to forget the closing character if you wait until after you've entered the contents.

## IntelliSense icons

When you type a period after the name of a class, IntelliSense displays the name of every member of that class. To the left of each member name is an icon that depicts the type of member. Common icons and their types include the following:

Icon	Meaning
	Method (discussed in Chapter 3)
	Class (discussed in Chapter 7)
	Struct (discussed in Chapter 9)
	Enum (discussed in Chapter 9)
	Extension method (discussed in Chapter 12, "Working with inheritance")
	Interface (discussed in Chapter 13, "Creating interfaces and defining abstract classes")
	Delegate (discussed in Chapter 17, "Introducing generics")
	Event (discussed in Chapter 17)
	Namespace (discussed in the next section of this chapter)

You'll also see other IntelliSense icons appear as you type code in different contexts.

## To build and run the console application

1. Open the **Build** menu and select **Build Solution**.

This compiles the C# code, resulting in a program that you can run. The Output window appears below the Code and Text Editor window.



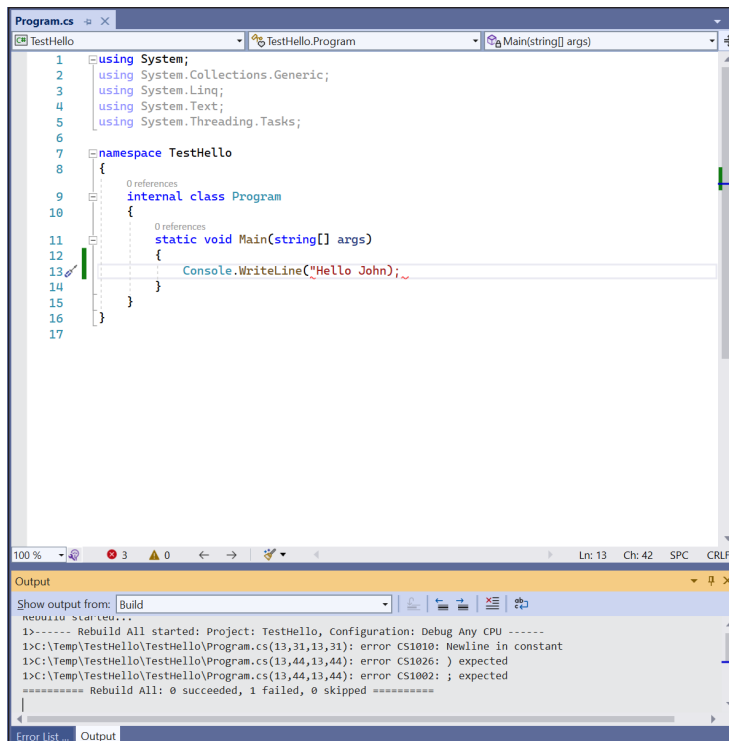
**Tip** If the Output window does not appear, select Output on the View menu to display it.

In the Output window, you should see messages similar to the following, indicating how the program is being compiled:

Build started...

```
1>----- Build started: Project: TestHello, Configuration: Debug Any CPU -----
1>TestHello -> C:\Users\student\Documents\Microsoft Press\VCSBS\Chapter 1\TestHello\
TestHello\bin\Debug\net6.0\TestHello.dll
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

If you made any mistakes, they'll be reported in the Error List window. The following image shows what happens if you forget to type the closing quotation marks after the text `Hello` in the `WriteLine` statement. Notice that a single mistake can sometimes cause multiple compiler errors.





**Tip** Notice that Visual Studio displays a wavy red line under any lines of code that will not compile when you enter them. To go directly to the line that caused the error, double-click an item in the Error List window.

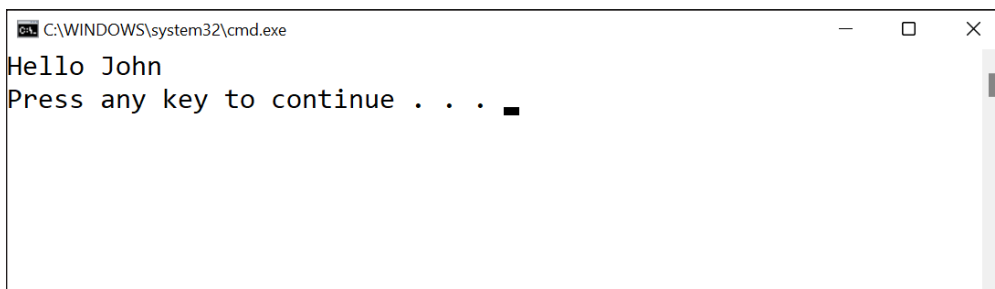
If you've followed the previous instructions carefully, there should be no errors or warnings, and the program should build successfully.



**Tip** There's no need to save the file explicitly before building because the Build Solution command automatically saves it. An asterisk after the file name in the tab above the Code and Text Editor window indicates that the file has been changed since it was last saved.

2. On the **Debug** menu, select **Start Without Debugging**.

A command window opens, and the program runs. The messages "Hello World!" and "Hello *YourName*" appear. The program now waits for you to press any key.



```
C:\WINDOWS\system32\cmd.exe
Hello John
Press any key to continue . . .
```



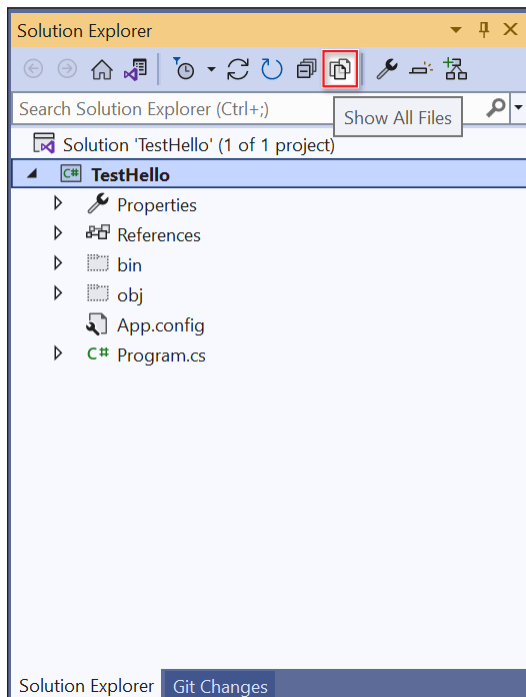
**Note** The "Press any key to continue" prompt is generated by Visual Studio. You didn't write any code to do this. If you run the program by using the Start Debugging command on the Debug menu, the application runs, but the command window closes immediately without waiting for you to press a key.

3. Ensure that the command window displaying the program's output is currently active, and press **Enter**.

The command window closes, and you return to the Visual Studio 2022 programming environment.



4. In Solution Explorer, select the **TestHello** project (not the solution). Then, on the Solution Explorer toolbar, select the **Show All Files** button.

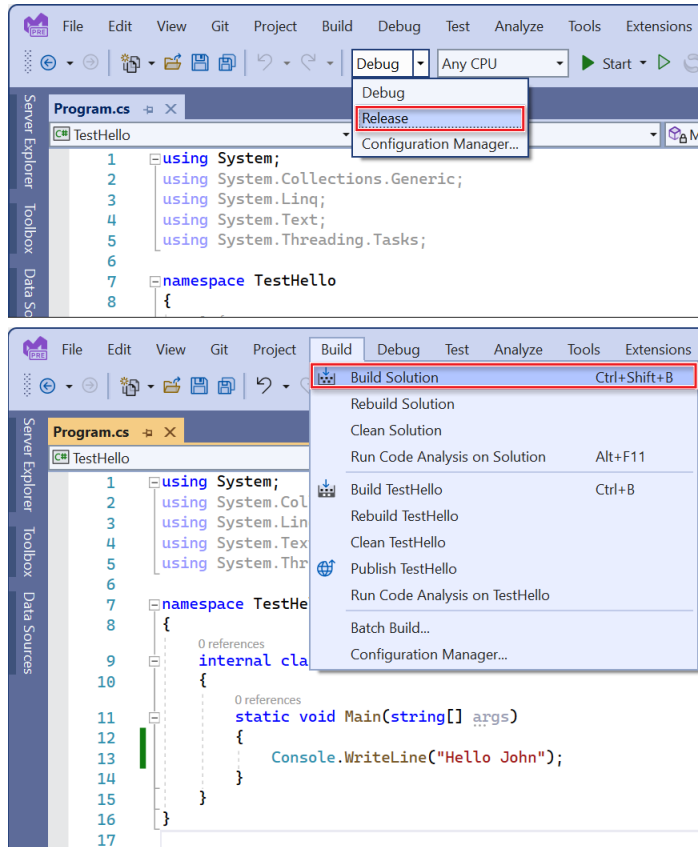


Entries named *bin* and *obj* appear above the *Program.cs* file. These entries correspond directly to folders named *bin* and *obj* in the project folder (Microsoft Press\VC SBS\Chapter 1\TestHello\TestHello). Visual Studio creates these folders when you build your application; they contain the executable version of the program and some other files used to build and debug the application.

5. In Solution Explorer, expand the **bin** entry.
6. In Solution Explorer, expand the **Debug** folder.

Several more items appear, including a file named *TestHello.exe*. This is the compiled program, which is the file that executes the program in debug mode. The other files contain information used by Visual Studio 2022 if you run your program in debug mode (when you select Start Debugging on the Debug menu).

You can also create a release build of an application. This build doesn't include the debugging information and is more compact (and probably quicker when it runs). When you've successfully built and tested an application, you build a release version and deploy it to your users. You create a release version by selecting the Release configuration in the Visual Studio toolbar and then using the Build Solution command from the Build menu.



## Using namespaces

The example you've seen so far is a very small program. However, small programs can soon grow into much bigger programs. As a program grows, two issues arise:

- It's harder to understand and maintain big programs than it is to understand and maintain smaller ones.
- More code usually means more classes, with more methods, requiring you to keep track of more names. As the number of names increases, so does the likelihood that the project build will fail because two or more names clash. For example, you might try to create two classes with the same name. The situation becomes more complicated when a program references assemblies written by other developers who have also used a variety of names.

In the past, programmers tried to solve the name-clashing problem by prefixing names with some sort of qualifier or set of qualifiers. Using prefixes as qualifiers is not a good solution, however, because it's not scalable. Names become longer, you spend less time writing software and more time typing (there is a difference), and you spend too much time reading and rereading incomprehensibly long names.

Namespaces help solve this problem by creating a container for items such as classes. Two classes with the same name won't be confused with each other if they live in different namespaces. You can create a class named `Greeting` inside the namespace named `TestHello` by using the namespace keyword like this:

```
namespace TestHello
{
 class Greeting
 {
 ...
 }
}
```

You can then refer to the `Greeting` class as `TestHello.Greeting` in your programs. If another developer also creates a `Greeting` class in a different namespace, such as `NewNamespace`, and you install the assembly that contains this class on your computer, your programs will still work as expected because they're using your `TestHello.Greeting` class. If you want to refer to the other developer's `Greeting` class, you must specify it as `NewNamespace.Greeting`.

It's good practice to define all your classes in namespaces, and the Visual Studio 2022 environment follows this recommendation by using the name of your project as the top-level namespace. The libraries provided with .NET Core and the .NET Framework also adhere to this practice; every class lives within a namespace. You've seen this approach in action already. The `Console` class lives in the `System` namespace, and its full name is actually `System.Console`. Indeed, in the first application you created at the start of this chapter, you used this fully qualified name to refer to the `Console` class. Of course, if you had to write the full name of a class every time you used it, the situation would be no better than prefixing qualifiers or even just naming the class with some globally unique name such as `SystemConsole`. Fortunately, you can solve this problem with a `using` directive in your programs. If you return to the `TestHello` program in Visual Studio 2022 and look at the file `Program.cs` in the Code and Text Editor window, you'll notice the following line at the top of the file:

```
using System;
```

This line is a `using` directive. A `using` directive brings a namespace into scope, enabling an application to use the items defined in the namespace. In the subsequent code in the same file, you no longer need to explicitly qualify objects in the `System` namespace. The `System` namespace contains classes that are used so often that Visual Studio 2022 automatically adds this `using` directive every time you create a new console project. When you create other types of projects, you may see additional namespaces referenced. You can add more `using` directives to the top of a source file if you need to reference other namespaces.



**Note** Sometimes `using` directives appear grayed-out. These directives correspond to namespaces that are not currently used by your application. If you don't need them when you've finished writing your code, you can safely delete them. However, if you require items that are held in these namespaces later, you will have to add the `using` directives back in again.

# Namespaces and assemblies

A `using` directive simply brings the items in a namespace into scope and frees you from having to fully qualify the names of classes in your code. Classes are compiled into assemblies. An assembly is a file that usually has the `.dll` file name extension, although strictly speaking, executable programs with the `.exe` file name extension are also assemblies.

An assembly can contain many classes. Classes such as `System.Console` are provided in assemblies that are installed on your computer together with Visual Studio. Class libraries provided with .NET Core and the .NET Framework contains thousands of classes. If they were all held in the same assembly, the assembly would be huge and difficult to maintain. (If Microsoft were to update a single method in a single class, it would have to distribute the entire class library to all developers!)

For this reason, the class library is split into a number of assemblies, partitioned by the functions they perform or the technology they implement. For example, .NET Core uses a “core” assembly named `System.Runtime.dll` that contains all the common C# types. Other assemblies contain classes for writing data to the console (such as `System.Console`), manipulating databases, accessing web services, building GUIs, and so on.

If you want to use a class in an assembly, you must add a reference to that assembly to your project. You can then add `using` directives to your code that bring the items in namespaces in that assembly into scope and make them accessible in your code. When you use Visual Studio to create an application, the template you select automatically includes references to the appropriate assemblies. To add references for additional assemblies to a .NET Core project, you use the NuGet Package Manager in Visual Studio. You’ll see this in action in later chapters.



**Note** The .NET Framework (as opposed to .NET Core) has its own “core” assembly, named `mscorlib.dll`, which contains many classes and methods that appear similar to the ones in `System.Runtime.dll`. However, these two assemblies are different and are not interchangeable. The classes and methods in `mscorlib` are Windows-specific implementations, whereas those in `System.Runtime.dll` are cross-platform—designed to work on Windows, Linux, and macOS. The `System.Runtime.dll` assembly doesn’t currently contain all the features available in `mscorlib`. However, you should not try to add the `mscorlib` assembly to a .NET Core application. If you require Windows-specific features, use a template based on the .NET Framework rather than .NET Core. If this all sounds technical, don’t worry. Microsoft is working hard to unify the two sets of libraries. The plan is that in the future, .NET Core and the .NET Framework will be replaced with a single runtime and set of assemblies simply named *.NET*.

There is not necessarily a 1:1 equivalence between an assembly and a namespace. A single assembly can contain classes defined in many namespaces, and a single namespace can span multiple assemblies. This all sounds very confusing at first, but you’ll soon get used to it.

## Commenting code

---

You will often see two forward slashes (//) followed by ordinary text in C# code. These are comments. They are ignored by the compiler but are very useful for developers because they help document what a program is actually doing. Take, for instance, the following example:

```
Console.ReadLine(); // Wait for the user to press the Enter key
```

The compiler skips all text from the two slashes to the end of the line. You can add multiline comments that start with a forward slash followed by an asterisk (/\*). The compiler skips everything until it finds an asterisk followed by a forward slash sequence (\*/), which could be many lines further down, as shown here:

```
/* This is a multi-line comment
All text is ignored by the compiler
until the closing comment sequence of characters */
```

You are encouraged to document your code with as many meaningful comments as necessary. You can also use comments to temporarily disable one or more lines of code, typically during testing or debugging.

## Creating a graphical application

---

So far, you've used Visual Studio 2022 to create and run a basic console application. The Visual Studio 2022 programming environment also contains everything you need to create graphical applications for Windows 10 and Windows 11. These templates are referred to as Universal Windows Platform (UWP) apps because they enable you to create apps that function on any device that runs Windows, such as desktop computers, tablets, and phones. You can design the user interface (UI) of a Windows application interactively. Visual Studio 2022 then generates the program statements to implement the user interface you've designed.

Visual Studio 2022 provides you with two views of a graphical application: the Code and Text Editor window, which you use to modify and maintain the code and program logic for a graphical application, and the Design View window, which you use to lay out your UI. You can switch between the two views whenever you want.

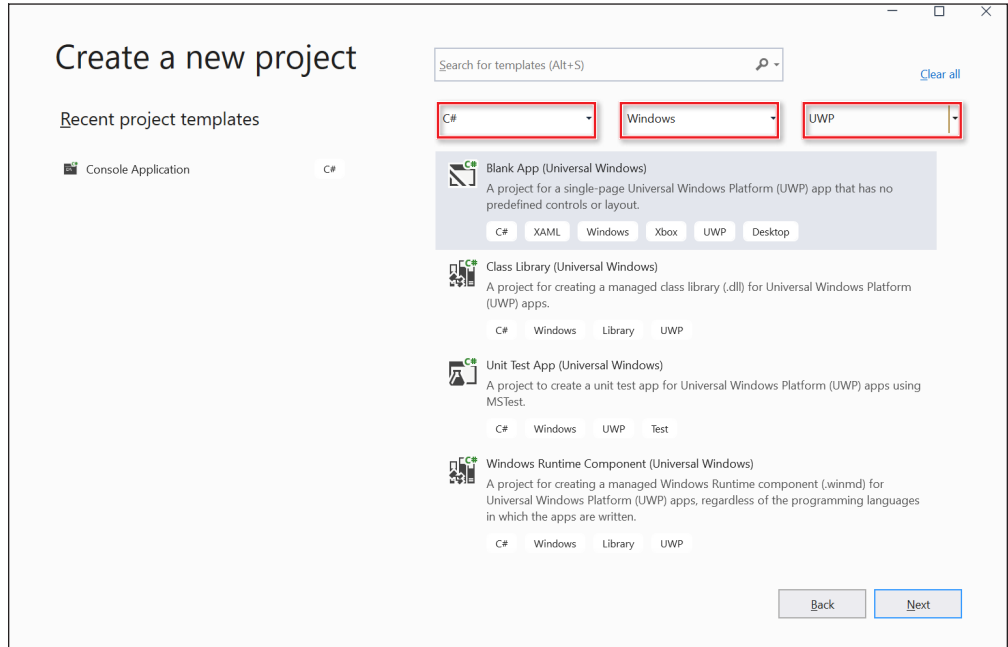
In the following set of exercises, you'll learn how to create a graphical application by using Visual Studio 2022. This program will display a simple form containing a box where you can enter your name and a button that, when selected, displays a personalized greeting in a message box.



**Note** If you want more information about the specifics of writing UWP apps, the chapters in Part IV of this book provide much more detail and guidance.

## To create a graphical application in Visual Studio 2022

1. If Visual Studio 2022 is still open from the previous exercise, open the **File** menu, choose **New**, and select **Project**. If you have closed Visual Studio, start it and select **Create a new project**.
2. In the Create a new project dialog, select **C#** in the **Language** dropdown list box, select **Windows** in the **Platform** list box, and select **UWP** in the **Project types** dropdown list box. Select the **Blank App (Universal Windows)** template, and then select **Next**.



3. In the Configure your new project dialog, enter the following values and select **Create**:
  - a. Set the project name to **HelloUWP**.
  - b. In the **Location** field, type **C:\Users\YourName\Documents\Microsoft Press\VCSBS\Chapter 1\**.
  - c. In the Solution dropdown list box, select **Create new solution**.

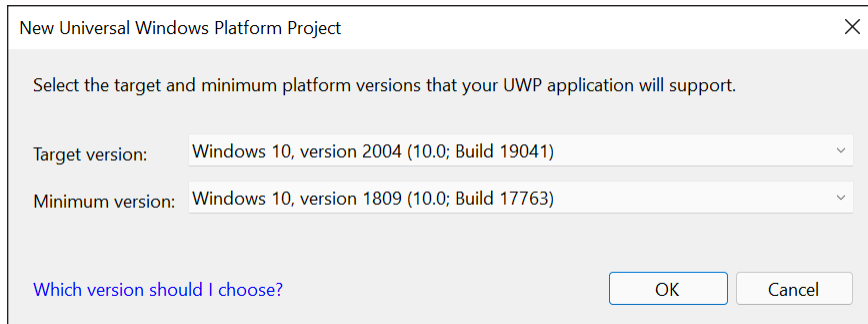


**Note** The Solution dropdown list box appears only if you were previously editing another application in Visual Studio 2022. It isn't displayed if you have just restarted Visual Studio.

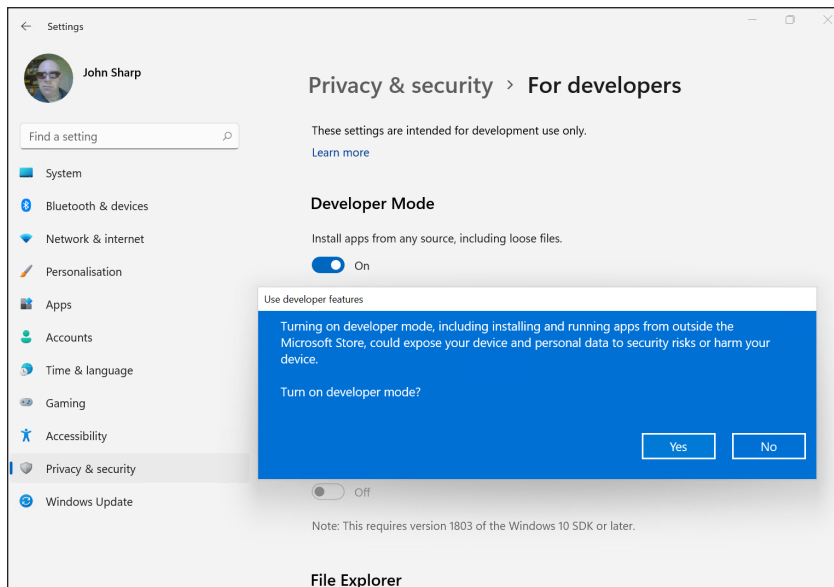
- d. Leave the **Solution name** box set to **HelloUWP**.
- e. Ensure that the **Place solution and project in the same directory** checkbox is left unchecked.

A dialog prompts you to specify which builds of Windows your application will run. Later builds of Windows have more and newer features available. Microsoft recommends that you always select the latest build of Windows as the target version, but if you're developing enterprise applications that also need to run on older versions, select the oldest version of Windows currently employed by users as the minimum version. However, don't automatically select the oldest version of Windows, as this might restrict some of the functionality available to your application.

4. Accept the default values. (These may vary from those shown in the following image, depending on which Windows updates you have applied to your computer.)



5. If this is the first time that you've created a UWP application, you might also be prompted to enable developer mode for Windows, and the Windows settings screen will appear. Select **Developer Mode**.
6. A dialog appears, confirming that this is what you want to do, as it bypasses some Windows security features. Select **Yes**. Windows downloads and installs the Developer Mode package, which provides additional features for debugging UWP applications.



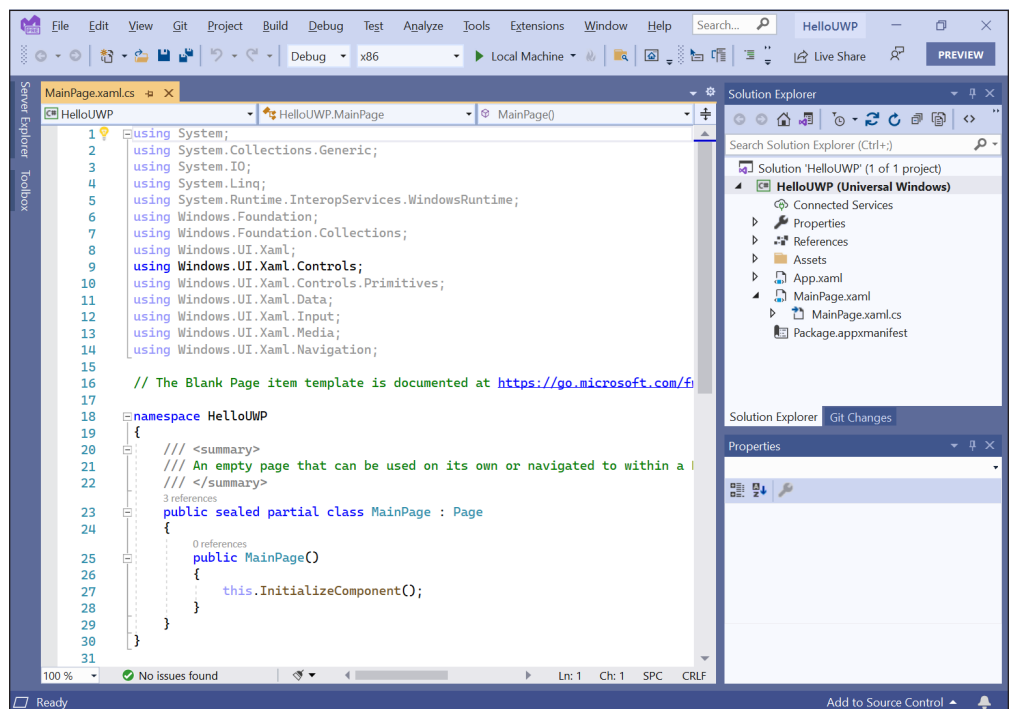


**Note** External apps that are not downloaded from the Windows Store could potentially expose personal data and pose other security risks, but it's necessary to enable Developer Mode if you are building and testing your own custom applications.

7. Return to Visual Studio. After the app has been created, look in the Solution Explorer pane.

Don't be fooled by the name of the application template. Although it's called Blank App, this template actually provides several files and contains some code. For example, if you expand the MainPage.xaml folder, you'll find a C# file named MainPage.xaml.cs.

8. Select this file to display it in the Code and Text Editor window. This file is where you'll add the initial code for the application.



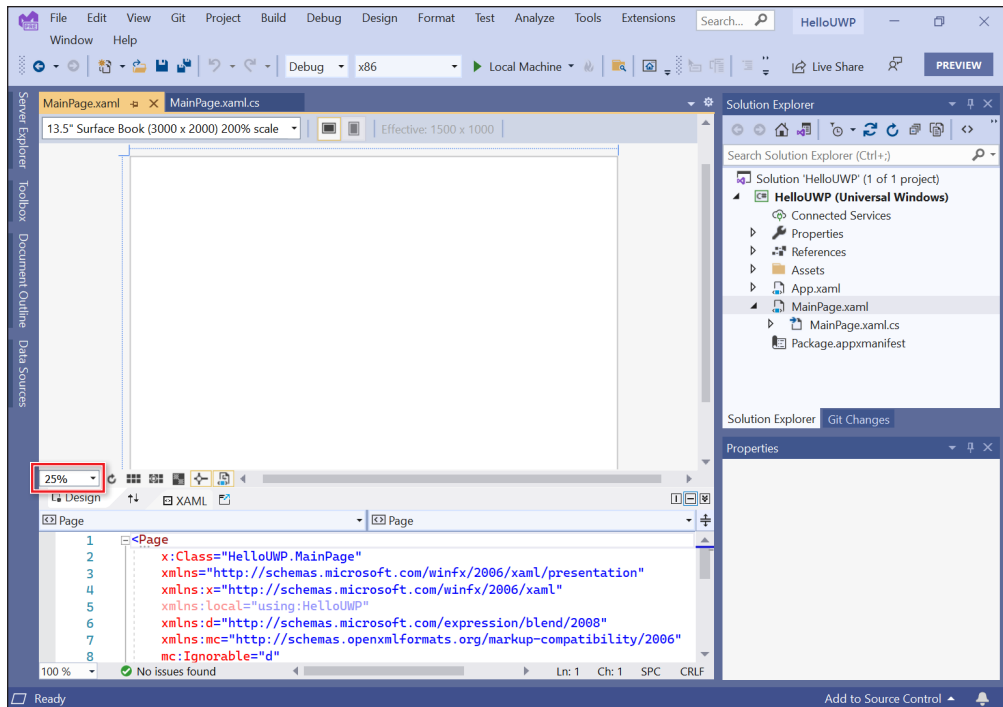
9. In the Solution Explorer pane, double-click **MainPage.xaml**.

This file contains the layout of the UI. The Design View window shows two representations of this file.



**Tip** Close the Output and Error List windows at the bottom of the screen to provide more space for displaying the Design View window.





At the top is a graphical view depicting the screen of a Surface Book, by default. You can use the percentage dropdown list box (highlighted in the preceding screen shot) to zoom in and out. The lower pane contains a description of the contents of this screen using XAML. XAML is an XML-like language used by UWP applications to define the layout of a form and its contents. If you have knowledge of XML, XAML should look familiar.



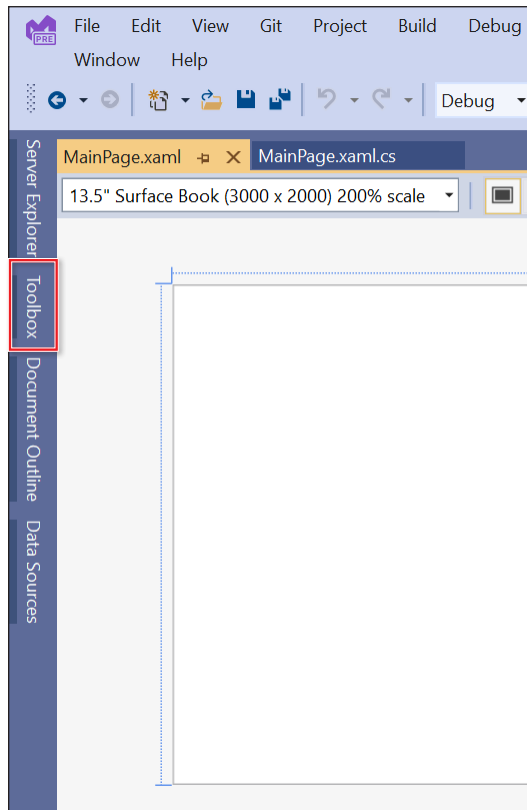
**Note** XAML stands for Extensible Application Markup Language, which is the language that UWP applications use to define the layout for the GUI of an application. You'll learn more about XAML as you progress through the exercises in this book.

Before going further, it's worth explaining some terminology. In traditional Windows applications, the UI consists of one or more *windows*, but in a UWP app, the corresponding items are referred to as *pages*. For the sake of clarity, I will simply refer to both items by using the blanket term *form*. However, I'll continue to use the word *window* to refer to items in the Visual Studio 2022 IDE, such as the Design View window.

In the following exercises, you'll use the Design View window to add three controls to the form displayed by your application. You will also examine some of the C# code automatically generated by Visual Studio 2022 to implement these controls.

## To create the user interface

1. Select the **Toolbox** tab in the margin to the left of the form in the Design View window.



The Toolbox appears, displaying the various components and controls that you can place on a form. By default, the General section of the toolbox is selected. It doesn't contain any controls (yet).

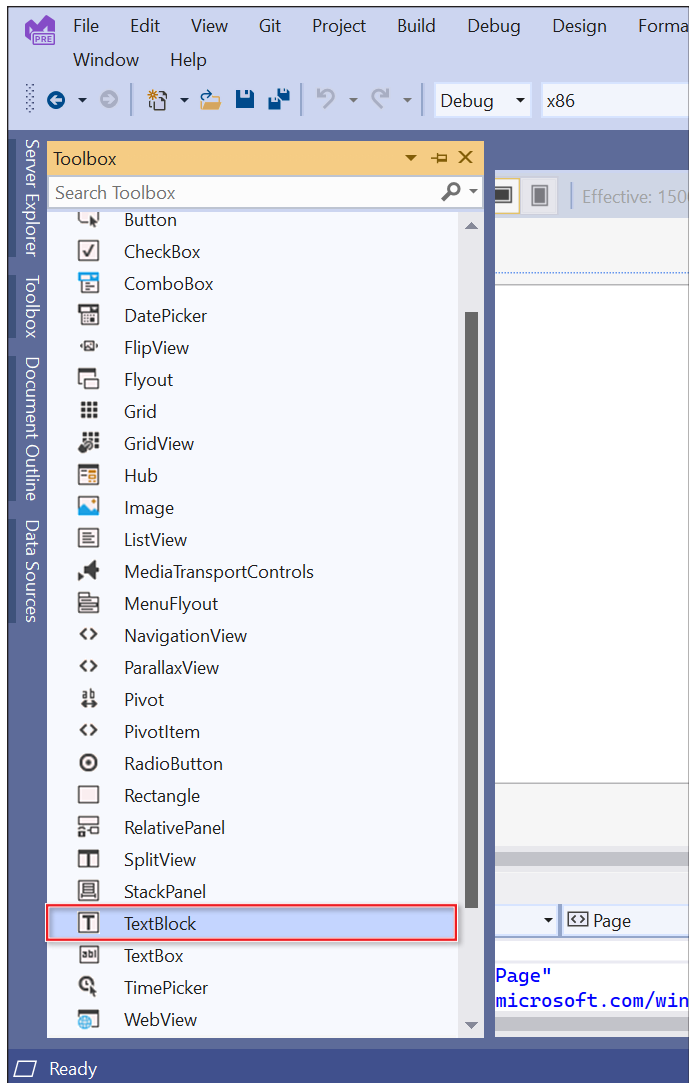
2. Expand the **Common XAML Controls** section.

This section displays a list of controls that most graphical applications use.



**Tip** The All XAML Controls section displays a more extensive list of controls.

3. In the Common XAML Controls section, select **TextBlock**. Then drag the **TextBlock** control onto the form displayed in the Design View window.



**Tip** Be sure to select the *TextBlock* control and not the *TextBox* control. If you accidentally place the wrong control on a form, you can easily remove it by selecting the item on the form and pressing Delete.

A *TextBlock* control is added to the form (you will move it to its correct location in a moment), and the Toolbox disappears from view.



**Tip** If you want the Toolbox to remain visible but not hide any part of the form, select the Auto Hide button (it looks like a pin) at the right end of the Toolbox title bar. The Toolbox is docked on the left side of the Visual Studio 2022 window, and the Design View window shrinks to accommodate it. (You might lose a lot of space if you have a low-resolution screen.) Selecting the Auto Hide button once more causes the Toolbox to disappear again.

4. The *TextBlock* control on the form is probably not exactly where you want it. You can select and drag the controls you've added to a form to reposition them. Using this technique, move the *TextBlock* control so it's positioned toward the upper-left corner of the form. (The exact placement is not critical for this application.) You might need to select away from the control and then select it again before you can move it in the Design View window.

The XAML description of the form in the lower pane now includes the *TextBlock* control, with properties such as its location on the form, governed by the *Margin* property, the default text displayed by this control in the *Text* property, the alignment of text displayed by this control as specified by the *HorizontalAlignment* and *VerticalAlignment* properties, and whether text should wrap if it exceeds the width of the control.

Your XAML code for the *TextBlock* will look similar to this (your values for the *Margin* property might be slightly different, depending on where you positioned the *TextBlock* control on the form):

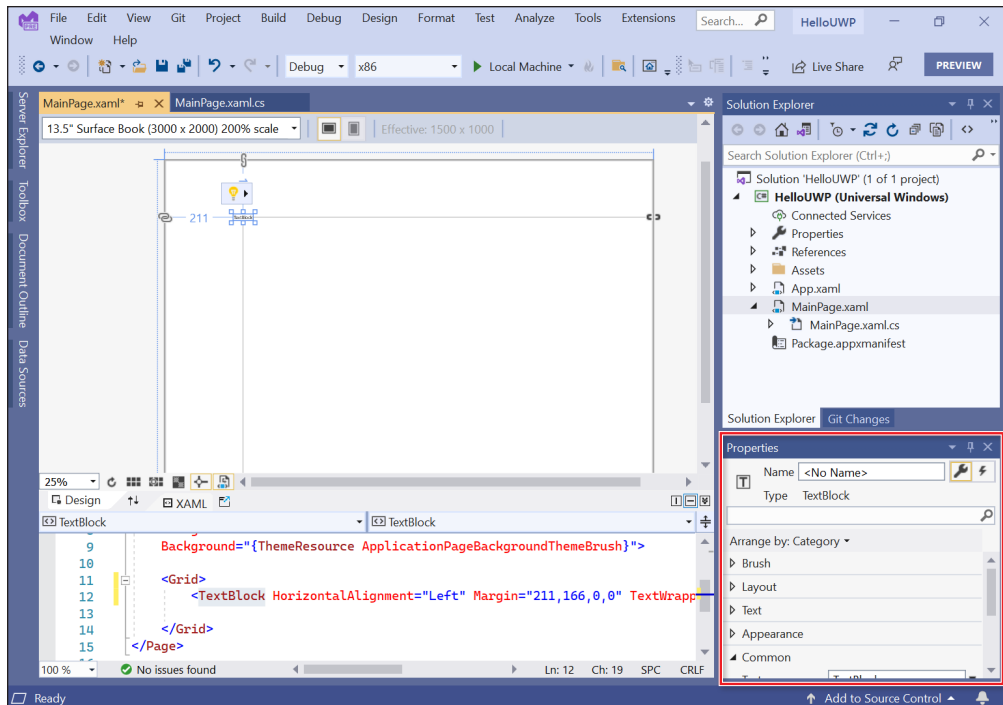
```
<TextBlock HorizontalAlignment="Left" Margin="211,166,0,0" TextWrapping="Wrap"
Text="TextBlock" VerticalAlignment="Top"/>
```

The XAML pane and the Design View window have a two-way relationship with each other. You can edit the values in the XAML pane, and the changes will be reflected in the Design View window. For example, you can change the location of the *TextBlock* control by modifying the values in the *Margin* property.

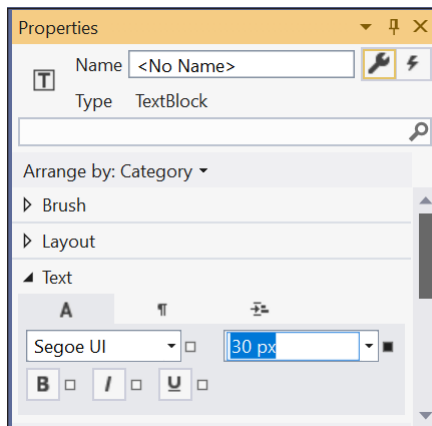
5. On the **View** menu, select **Properties Window**. (It's the penultimate item in the menu.)

If it wasn't already displayed, the Properties window appears at the lower-right area of the screen, under the Solution Explorer pane. You can specify the properties of controls by using the XAML pane under the Design View window, but the Properties window provides a more convenient way to modify the properties for items on a form as well as other items in a project.

The Properties window is context sensitive, in that it displays the properties for the currently selected item. If you select the form displayed in the Design View window (outside the *TextBlock* control), you can see that the Properties window displays the properties for a *Grid* element. If you look at the XAML pane, you should see that the *TextBlock* control is contained within a *Grid* element. All forms contain a *Grid* element that controls the layout of displayed items; for example, you can define tabular layouts by adding rows and columns to the *Grid* element.



6. In the Design View window, select the **TextBlock** control. The Properties window displays the properties for the TextBlock control.
7. In the Properties window, expand the **Text** property of the **TextBlock** control. Change the **FontSize** property to **30 px** and press **Enter**. This property is located next to the dropdown list box containing the name of the font, which will show Segoe UI.



**Note** The suffix *px* indicates that the font size is measured in pixels.

8. In the XAML pane below the Design View window, examine the text that defines the *TextBlock* control. If you scroll to the end of the list, you should see the text `FontSize="30"`. Any changes you make using the Properties window are automatically reflected in the XAML definitions, and vice versa.

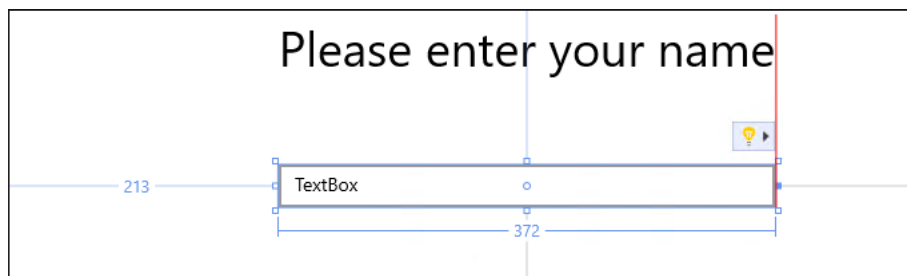
Type over the value of the *FontSize* attribute in the XAML pane and change it to **36**. The font size of the text for the *TextBlock* control in the Design View window and the Properties window changes.

9. In the Properties window, examine the other properties of the *TextBlock* control. Feel free to experiment by changing them to see their effects. Notice that as you change the values of properties, these properties are added to the definition of the *TextBlock* control in the XAML pane. Each control that you add to a form has a default set of property values, and these values are not displayed in the XAML pane unless you change them.
10. Change the value of the **Text** property of the *TextBlock* control from **TextBlock** to **Please enter your name**. You can do this by editing the Text element in the XAML pane or by changing the value in the Properties window. (This property is located in the Common section in the Properties window.)
11. Select the form in the Design View window, and then display the Toolbox again.
12. In the Toolbox, select and drag the **TextBox** control onto the form. Move the *TextBox* control so that it's directly below the *TextBlock* control.

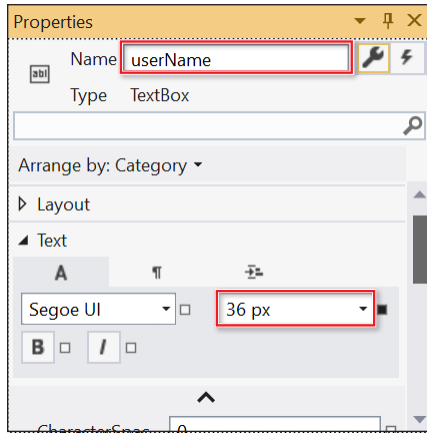


**Tip** When you drag a control on a form, alignment indicators appear automatically. These give you a quick visual cue to ensure that controls are lined up neatly. You can also manually edit the `Margin` property in the XAML pane to set the left margin to the same value of that for the *TextBlock* control.

13. In the Design View window, hover the mouse pointer over the handle on the right edge of the **TextBox** control. The mouse pointer changes to a double-headed arrow, indicating that you can resize the control. Drag the right edge of the **TextBox** control until it is aligned with the right edge of the *TextBlock* control above. If you can't see the handle for the right edge, use the percentage dropdown list box control to zoom in.

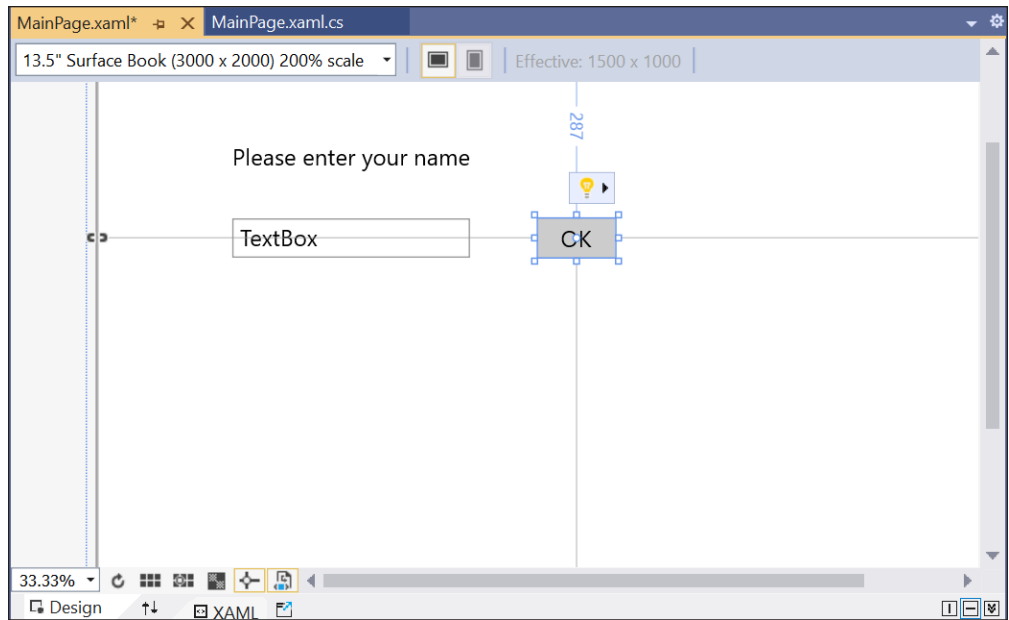


14. While the *TextBox* control is selected, at the top of the Properties window, change the value of the **Name** property from **<No Name>** to **userName**. Also change the font size of the **Text** property to **36 px**, to match the *TextBlock* control.



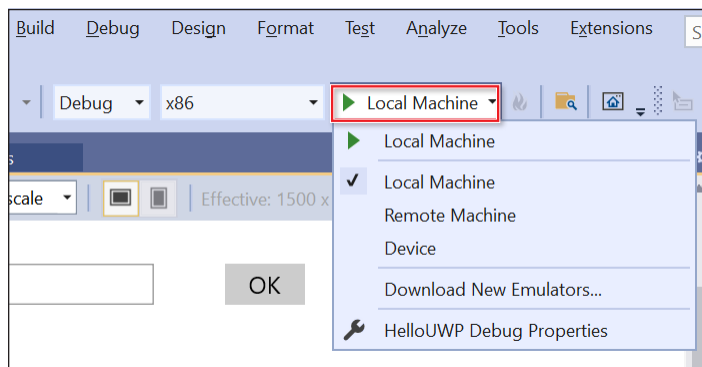
**Note** You'll learn more about naming conventions for controls and variables in Chapter 2, "Working with variables, operators, and expressions."

15. Display the Toolbox again. Then select and drag a **Button** control onto the form, to the right of the *TextBox* control on the form, so the bottom of the button is aligned horizontally with the bottom of the box.
16. In the Properties window, change the **Name** property of the **Button** control to **ok**, set the font size to **36 px**, and change the **Content** property (in the Common section in the Properties window) from **Button** to **OK**, and press **Enter**.
17. Verify that the caption of the *Button* control on the form changes to display the text **OK**.
18. In the Layout section in the Properties window, change the width to **120**. The form should now look similar to the following figure.



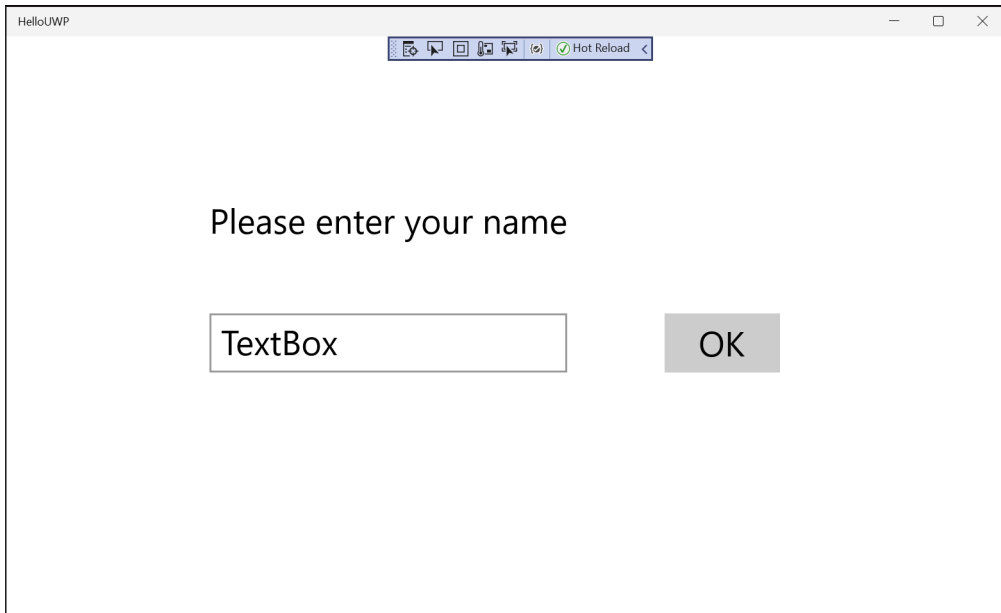
**Note** The dropdown menu in the upper-left corner of the Design View window enables you to view how your form will render on different screen sizes and resolutions. In this example, the default view of a 13.5-inch Surface Book with a 3,000 x 2,000 resolution is selected. To the right of the dropdown menu, two buttons enable you to switch between portrait view and landscape view.

19. On the **Build** menu, select **Build Solution**, and then verify that the project builds successfully.
20. Ensure that the **Debug Target** dropdown list is set to **Local Machine**. (It might default to Device and attempt to connect to a Windows phone device, in which case the build will probably fail.) Then, on the **Debug** menu, select **Start Debugging**.





The application should run and display your form. The form looks like this:



**Note** When you run a UWP app in debug mode, a debug toolbar appears near the top of the form. You can use this toolbar to track how the user is navigating through the form and monitor how the contents of the controls on the form change. You can ignore this menu for now; select the double bar at the bottom of the toolbar to minimize it.

21. In the box, overtype the existing text with your name, and then select **OK**.

Nothing happens yet. You need to add some code to indicate what should happen when the user selects the OK button, which is what you'll do next.

22. Return to Visual Studio 2022. On the **Debug** menu, select **Stop Debugging**.



**Note** You can also select the Close button (the X in the upper-right corner of the form) to close the form, stop debugging, and return to Visual Studio.

You've managed to create a graphical application without writing a single line of C# code. It doesn't do much yet (you will have to write some code soon), but Visual Studio 2022 generates a lot of code to handle routine tasks that all graphical applications must perform, such as starting up and displaying a window. Before adding your own code to the application, it helps to understand what Visual Studio has produced for you. The following section describes these automatically generated artifacts.

## Examining the Universal Windows Platform app

In Solution Explorer, expand the MainPage.xaml node. The file MainPage.xaml.cs appears; double-click this file. The following code for the form is displayed in the Code and Text Editor window:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// The Blank Page item template is documented at https://go.microsoft.com/fwlink/?LinkId=402352
// &clcid=0x409

namespace HelloUWP
{
 /// <summary>
 /// An empty page that can be used on its own or navigated to within a Frame.
 /// </summary>
 public sealed partial class MainPage : Page
 {
 public MainPage()
 {
 this.InitializeComponent();
 }
 }
}
```

In addition to a good number of `using` directives to bring into scope some namespaces that most UWP apps use, the file contains the definition of a class called `MainPage` but not much else. There's a little bit of code for the `MainPage` class known as a constructor that calls a method named `InitializeComponent`. A constructor is a special method with the same name as the class. It runs when an instance of the class is created and can contain code to initialize the instance. You'll learn about constructors in Chapter 7.

The class contains a lot more code than the few lines shown in the `MainPage.xaml.cs` file, but much of it is generated automatically based on the XAML description of the form and is hidden from you. This hidden code performs operations such as creating and displaying the form and creating and positioning the various controls on the form.

At this point, you might be wondering where the `Main` method is and how the form gets displayed when the application runs. Remember that in a console application, `Main` defines the point at which the program starts. A graphical application is slightly different.

In Solution Explorer, notice another source file: App.xaml. If you expand the node for this file, you'll see another file called App.xaml.cs. In a UWP app, the App.xaml file provides the entry point at which the application starts running. If you double-click App.xaml.cs in Solution Explorer, you should see some code that looks similar to this:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.ApplicationModel;
using Windows.ApplicationModel.Activation;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

namespace HelloUWP
{
 /// <summary>
 /// Provides application-specific behavior to supplement the default Application class.
 /// </summary>
 sealed partial class App : Application
 {
 /// <summary>
 /// Initializes the singleton application object. This is the first line of authored code
 /// executed, and as such is the logical equivalent of main() or WinMain().
 /// </summary>
 public App()
 {
 this.InitializeComponent();
 this.Suspending += OnSuspending;
 }

 /// <summary>
 /// Invoked when the application is launched normally by the end user. Other entry points
 /// will be used such as when the application is launched to open a specific file.
 /// </summary>
 /// <param name="e">Details about the launch request and process.</param>
 protected override void OnLaunched(LaunchActivatedEventArgs e)
 {
 Frame rootFrame = Window.Current.Content as Frame;

 // Do not repeat app initialization when the Window already has content,
 // just ensure that the window is active
 if (rootFrame == null)
 {
 // Create a Frame to act as the navigation context and navigate to the first page
 rootFrame = new Frame();

 rootFrame.NavigationFailed += OnNavigationFailed;
 }
 }
 }
}
```

```

 if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
 {
 // TODO: Load state from previously suspended application
 }

 // Place the frame in the current Window
 Window.Current.Content = rootFrame;
 }

 if (e.PrelaunchActivated == false)
 {
 if (rootFrame.Content == null)
 {
 // When the navigation stack isn't restored navigate to the first page,
 // configuring the new page by passing required information as a navigation
 // parameter
 rootFrame.Navigate(typeof(MainPage), e.Arguments);
 }
 // Ensure the current window is active
 Window.Current.Activate();
 }
}

/// <summary>
/// Invoked when Navigation to a certain page fails
/// </summary>
/// <param name="sender">The Frame which failed navigation</param>
/// <param name="e">Details about the navigation failure</param>
void OnNavigationFailed(object sender, NavigationFailedEventArgs e)
{
 throw new Exception("Failed to load Page " + e.SourcePageType.FullName);
}

/// <summary>
/// Invoked when application execution is being suspended. Application state is saved
/// without knowing whether the application will be terminated or resumed with the contents
/// of memory still intact.
/// </summary>
/// <param name="sender">The source of the suspend request.</param>
/// <param name="e">Details about the suspend request.</param>
private void OnSuspending(object sender, SuspendingEventArgs e)
{
 var deferral = e.SuspendingOperation.GetDeferral();
 //TODO: Save application state and stop any background activity
 deferral.Complete();
}
}
}

```

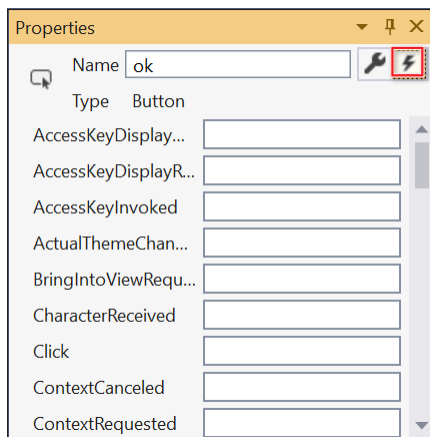
Much of this code consists of comments (the lines beginning `///`) and other statements that you don't need to understand just yet, but the key elements are located in the `OnLaunched` method, highlighted in bold. This method runs when the application starts, and the code in this method causes the application to create a new `Frame` object, display the `MainPage` form in this frame, and then activate it. It's not necessary at this stage to fully comprehend how this code works or the syntax of any of these statements, but it's helpful to simply appreciate that this is how the application displays the form when it starts running.

## Adding code to the graphical application

Now that you know a little bit about the structure of a graphical application, the time has come to write some code to make your application actually do something. In the next exercise, you'll write C# code that runs when the user selects the OK button. This code will display a personalized greeting, using the name entered by the user in the TextBox control.

### To write the code for the OK button

1. Double-click the **MainPage.xaml** file in Solution Explorer to open the form in the Design View window.
2. In the Design View window, select the **OK** button on the form to select it.
3. In the Properties window, select the **Event Handlers** button. This button displays an icon that looks like a bolt of lightning.



The Properties window displays a list of event names for the *Button* control. An event indicates a significant action that usually requires a response, such as the user selecting the button. You can write your own code to perform this response.

4. In the box adjacent to the Click event, type **okClick**, and press **Enter**.

The `MainPage.xaml.cs` file appears in the Code and Text Editor window, and a new method named `okClick` is added to the `MainPage` class. The method looks like this:

```
private void okClick(object sender, RoutedEventArgs e)
{
}
}
```

Don't worry too much about the syntax of this code just yet; you will learn all about methods in Chapter 3.

5. Scroll up and add the following using directive shown in bold to the list at the top of the file (the ellipsis character [...] indicates statements that have been omitted for brevity):

```
using System;
...
using Windows.UI.Xaml.Navigation;
using Windows.UI.Popups;
```

6. Return to the `onClick` method and add the code shown here in bold:

```
private void onClick(object sender, RoutedEventArgs e)
{
 MessageDialog msg = new MessageDialog($"Hello {userName.Text}");
 _ = msg.ShowAsync(); // Display the message box, but ignore the value returned
}
```

This code will run when the user selects the OK button. Again, don't worry too much about the syntax. Just be sure you copy the code exactly as shown; you'll find out what these statements mean in the next few chapters. The key things to understand are that the first statement creates a `MessageDialog` object with the message "Hello <YourName>" where <YourName> is the name that you type into the `TextBox` control on the form. The second statement displays the `MessageDialog`, causing it to appear on the screen. The `MessageDialog` class is defined in the `Windows.UI.Popups` namespace, which is why you added it in step 5.



**Note** The `ShowAsync()` method returns a value, but in this example, you aren't interested in what this value is. You simply want to use the method to display the message. In cases such as this, the syntax `_ =` indicates that the value should be discarded. If you omit these two characters, the code will still work, but Visual Studio will display a warning. Visual Studio does this by adding a wavy green line under the entire statement. If you hover over this line of code, Visual Studio displays a warning message that states, "Because this call is not awaited, execution of the current method continues before the call is completed. Consider applying the 'await' operator to the result of the call. Expression value is never used." Essentially, this alerts you to the fact that your code has ignored the return value, but you might not be aware of this. In cases such as this, it's always better to be explicit and highlight that you're intentionally ignoring the return value. This syntax is called a *discard*.

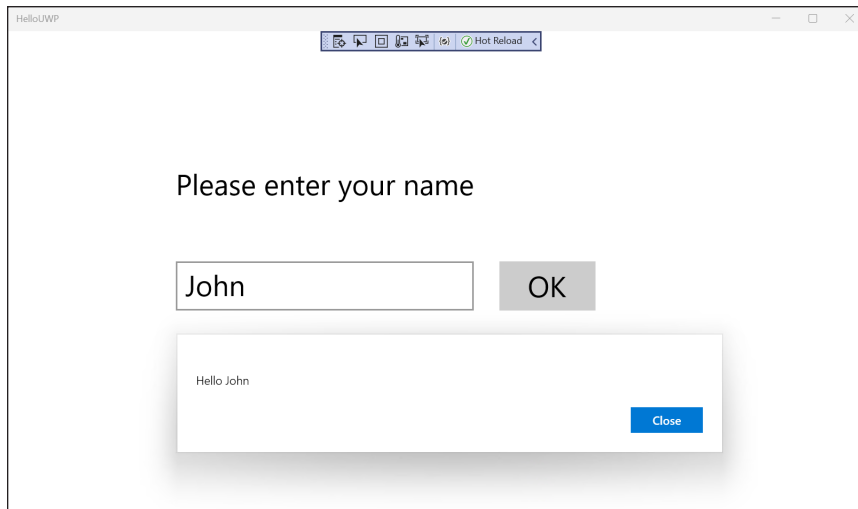
7. Select the **MainPage.xaml** tab above the Code and Text Editor window to display the form in the Design View window again.
8. In the lower pane displaying the XAML description of the form, examine the **Button** element, but be careful not to change anything. Notice that it now contains an attribute named `Click` that refers to the `onClick` method.

```
<Button x:Name="ok" ... Click="onClick" />
```

9. On the **Debug** menu, select **Start Debugging**.

10. When the form appears, in the box, replace the existing text with your name and then select **OK**.

A message dialog appears displaying the following greeting:



11. Select **Close** in the message box.
12. Return to Visual Studio 2022. Then, on the **Debug** menu, select **Stop Debugging**.

## Other types of graphical applications

In addition to UWP apps, Visual Studio 2022 lets you create other types of graphical applications. These applications are intended for specific environments and don't include the adaptability to enable them to run across multiple platforms unchanged.

The other types of graphical application templates available in Visual Studio 2022 include the following:

- **WinUI3 app** WinUI3 is the latest set of APIs and tooling for building graphical applications. A UWP app runs in a security-constrained sandbox, which limits some of the functionality available. A WinUI3 app runs as a native desktop app and has full access to all the features provided by Windows. You might reasonably ask, "Why does this book use UWP rather than WinUI3?" The answer is that, at the time of writing, WinUI3 is currently a work in progress and isn't fully complete. Using UWP provides a degree of stability that WinUI3 doesn't yet have, although I expect to switch all the code and samples in this book to WinUI3 if there's a future edition! Additionally, Microsoft has stated that the graphical design tools for WinUI3 won't be available as part of the first release, which means you'll have to develop the user interface manually—either by handcrafting your own XAML markup from scratch or by using various C# APIs. So, to keep things simple, I elected to stick to UWP for most of this book; it doesn't affect C#, just the way in which you create your UI.

- **.NET MAUI app** MAUI stands for Multi-platform Application User Interface. A .NET MAUI app can run on Windows machines and non-Windows devices. Operating systems supported include Android, macOS, iOS, and iPadOS. Using .NET MAUI, you can build mobile and desktop apps that can run unchanged across multiple platforms, such as a Microsoft Surface laptop, a Mac Mini, an iPhone, and an Android tablet. .NET MAUI is outside the scope of this book, but the essentials are very similar to building a UWP application. You define the user interface using XAML, and you write the application logic using C#. There are some differences in the controls available from a UWP app, and the structure of applications is slightly different due to the nature of the underlying operating systems and device capabilities, but once you know how to create UWP applications, it's relatively straightforward to move on to .NET MAUI apps.
- **WPF app** WPF stands for Windows Presentation Foundation. WPF is targeted at applications that run on the Windows desktop rather than applications that can adapt to a range of different devices and form factors. It provides an extremely powerful framework based on vector graphics that enable the user interface to scale smoothly across different screen resolutions. Many of the key features of WPF are available in UWP applications, although WPF provides additional functionality that is appropriate only for applications running on powerful desktop machines.
- **Windows Forms app** This is an older graphical library that dates back to the origins of the .NET Framework. As its name implies, the Windows Forms library is intended for building more classical forms-based applications using the Graphics Device Interface (GDI) libraries provided with Windows at that time. Although this framework is quick to use, it provides neither the functionality and scalability of WPF nor the portability of UWP.

If you're building graphical applications, unless you have good reasons not to do so, I suggest you opt for the UWP template.

## Summary

In this chapter, you saw how to use Visual Studio 2022 to create, build, and run applications. You created a console application that displays its output in a console window, and you created a UWP application with a simple GUI.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 2.
- If you want to exit Visual Studio 2022 now, open the File menu and select Exit. If Windows displays a Save dialog, select Yes to save the project.



## Quick Reference

To	Do this
Create a new console application using the .NET CLI	On the command line, run the following command to create the app: <code>dotnet new console</code> To build and run an app, use the following command: <code>dotnet new console</code>
Create a new console application using Visual Studio 2022	Open the File menu, point to New, and then select Project to open the Create a new project dialog. Select the C# language, the Windows platform, and the Console project type. Select the Console Application template. In the Configure your new project dialog, specify a project name, a location, and a solution name. In the Location box, specify a directory for the project files. In the Additional information dialog, select .NET 6.0 as the target framework.
Create a new UWP app using Visual Studio 2022	Open the File menu, choose New, and select Project to open the Create a new project dialog. Select the C# language, the Windows platform, and the UWP project type. Select the Blank App (Universal Windows) template. In the Configure your new project dialog, specify a project name, a location, and a solution name. In the Location box, specify a directory for the project files. Specify the version of Windows on which you want the application to be able to run.
Build the application	On the Build menu, select Build Solution.
Run the application in debug mode	On the Debug menu, select Start Debugging.
Run the application without debugging	On the Debug menu, select Start Without Debugging.

# Working with variables, operators, and expressions

**After completing this chapter, you will be able to:**

- Understand statements, identifiers, and keywords.
- Use variables to store information.
- Work with primitive data types.
- Use arithmetic operators such as the plus sign (+) and the minus sign (−).
- Increment and decrement variables.
- Declare implicitly typed local variables.

Chapter 1, "Welcome to C#," introduced how to use the Microsoft Visual Studio 2022 programming environment to build and run a console program and a graphical application. This chapter introduces you to Microsoft Visual C# syntax and semantics, including statements, keywords, and identifiers. You'll study the primitive types that are built into the C# language and the characteristics of the values that each type holds. You'll also see how to declare and use local variables (variables that exist only in a method or another small section of code), learn about the arithmetic operators that C# provides, find out how to use operators to manipulate values, and learn how to control expressions containing two or more operators.

## Understanding statements

---

A *statement* is a command that performs an action, such as calculating a value and storing the result or displaying a message to a user. You combine statements to create methods. You'll learn more about methods in Chapter 3, "Writing methods and applying scope," but for now, think of a method as a named sequence of statements. `Main`, which was introduced in Chapter 1, is an example of a method.

Statements in C# follow a well-defined set of rules describing their format and construction. These rules are collectively known as *syntax*. (In contrast, the specification of what statements *do* is collectively known as *semantics*.) One of the simplest and most important C# syntax rules states that you must terminate all statements with a semicolon. For example, Chapter 1 demonstrates that without the terminating semicolon, the following statement won't compile:

```
Console.WriteLine("Hello, World!");
```



**Tip** C# is a free format language, which means that white space, such as a space character or a new line, is not significant except as a separator. In other words, you are free to lay out your statements in any style you choose. However, you should adopt a simple, consistent layout style to make your programs easier to read and understand.

The trick to programming well in any language is to learn the syntax and semantics of the language and then use the language in a natural and idiomatic way. This approach makes your programs easier to maintain. (Code that's difficult to read is said to be *obfuscated*. This type of code may appear to be clever but is a real pain if you, or someone else, needs to come back to it later!) As you progress through this book, you'll see examples of the most important C# statements.

## Using identifiers

*Identifiers* are the names that you use to identify the elements in your programs, such as namespaces, classes, methods, and variables. (You'll learn about variables shortly.) In C#, you must adhere to the following syntax rules when choosing identifiers:

- You can use only letters (uppercase and lowercase), digits, and underscore characters.
- An identifier must start with a letter or an underscore.

For example, `result`, `_score`, `footballTeam`, and `plan9` are all valid identifiers, whereas `result%`, `footballTeam$`, and `9plan` are not.



**Important** C# is a case-sensitive language. For example, `footballTeam` and `FootballTeam` are two different identifiers.

## Identifying keywords

C# reserves 77 identifiers for its own use. You cannot reuse these identifiers for your own purposes. These identifiers are called *keywords*, and each has a particular meaning. Examples of keywords are

class, namespace, and using. You'll learn the meaning of most of the C# keywords as you proceed through this book. The following is the list of keywords:

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	



**Tip** In the Visual Studio 2022 Code and Text Editor window, keywords are colored blue when you type them.

C# also uses the following identifiers. These identifiers are not reserved by C#, which means you can use these names as identifiers for your own methods, variables, and classes, but you should avoid doing so if possible because it can cause confusion to another developer attempting to understand and maintain your code.

add	global	select
alias	group	remove
and	into	set
ascending	join	value
async	let	var
await	nameof	when
descending	not	where
dynamic	or	yield
from	orderby	
get	partial	

# Using variables

---

A *variable* is a storage location that holds a value. You can think of a variable as a box in the computer's memory that holds temporary information. You must give each variable in a program an unambiguous name that uniquely identifies it in the context in which it is used.

You use a variable's name to refer to the value it holds. For example, if you want to store the value of the cost of an item in a store, you might create a variable called `cost` and store the item's cost in this variable. Later, if you refer to the `cost` variable, the value retrieved will be the item's cost that you stored there earlier.

## Naming variables

You should adopt a naming convention for variables to help avoid confusion concerning the variables you have defined. This is especially important if you are part of a project team with several developers working on different parts of an application. A consistent naming convention helps to avoid confusion and can reduce the scope for bugs. The following list contains some general naming recommendations:

- Don't start an identifier with an underscore. Although this is legal in C#, it can limit the interoperability of your code with applications built by using other languages, such as Microsoft Visual Basic.
- Don't create identifiers that differ only by case. For example, don't create one variable named `myVariable` and another named `MyVariable` for use at the same time because it's too easy to confuse one with the other. Also, defining identifiers that differ only by case can limit the ability to reuse classes in applications developed with other languages that are not case sensitive, such as Visual Basic.
- Start the name with a lowercase letter.
- In a multi-word identifier, start the second and each subsequent word with an uppercase letter. (This is called *camelCase notation*.)
- Don't use Hungarian notation. (If you are a Microsoft Visual C++ developer, you're probably familiar with Hungarian notation. If you don't know what Hungarian notation is, don't worry about it!)

For example, `score`, `footballTeam`, `_score`, and `FootballTeam` are all valid variable names, but only the first two are recommended.

## Declaring variables

Variables hold values. C# has many different types of values that it can store and process: integers, floating-point numbers, and strings of characters, to name three. When you declare a variable, you must specify the type of data it will hold.

You declare the type and name of a variable in a declaration statement. For example, the statement that follows declares that the variable named `age` holds `int` (integer) values. As always, you must terminate the statement with a semicolon.

```
int age;
```

The variable type `int` is the name of one of the *primitive C# types*, *integer*, which is a whole number. (You'll learn about several primitive data types later in this chapter.)



**Note** If you're a Visual Basic programmer, you should note that C# does not allow implicit variable declarations. You must explicitly declare all variables before you use them.

After you've declared your variable, you can assign it a value. The statement that follows assigns `age` the value 42. Again, note that the semicolon is required.

```
age = 42;
```

The equal sign (=) is the *assignment* operator, which assigns the value on its right to the variable on its left. After this assignment, you can use the `age` variable in your code to refer to the value it holds. The next statement writes the value of the `age` variable (42) to the console:

```
Console.WriteLine(age);
```



**Tip** If you leave the mouse pointer over a variable in the Visual Studio 2022 Code and Text Editor window, a ScreenTip indicates the type of the variable.

## Specifying numeric values

It's important to understand the impact of variable type on the data that a variable can hold and how this data is handled. For example, it should be obvious that a numeric variable cannot hold a string value such as "Hello." However, in some cases, the type of a value being assigned to a variable is not always so clear-cut.

Take the literal value 42 as an example. It is numeric. Furthermore, it is an integer, and you can assign it directly to an integer variable. But what happens if you try to assign this value to a non-integer type, such as a floating-point variable? The answer is that C# will silently convert the integer value to a floating-point value. This is relatively harmless, but it's not necessarily good practice. You should specify that you intended to treat the literal value 42 as a floating-point number and haven't mistakenly assigned it to the wrong type of variable. You can do this by appending the `F` suffix to the numeric literal, like this:

```
float myVar; // declare a floating-point variable
myVar = 42F; // assign a floating-point value to the variable
```

How about the value 0.42? What is the type of this expression? The answer is that, like all numeric literals that include a decimal point, it is actually a double-precision floating-point number, referred to as a *double* for short. In the next section, you'll see that a double has a bigger range and greater precision than a float. If you want to assign the value 0.42 to a float, you must apply the F suffix (the C# compiler insists on this):

```
myVar = 0.42F;
```

C# has other numeric types: long integers, which have a bigger range than integers, and decimals, which hold exact decimal values. (Floats and doubles can be subject to rounding and other approximations when they are involved in calculations.) You should use the L suffix to assign a numeric literal value to a long integer, and the M suffix to assign a numeric literal value to a decimal.

This might seem to be a trivial point, but it is surprising how many subtle errors can creep into a program by accidentally assigning a value of the wrong type to a variable. Consider what might happen if you attempt to perform a calculation that involves a large number of significant decimal places and store the results in a float. In the worst case, this could lead to truncating data, causing errors in calculations. For example, if your program was a guidance application for a space-probe, such errors could cause it to completely miss Mars and to head off to the deeper depths of the solar system instead!

## Working with primitive data types

The numeric types of C#, together with other common types for holding strings, characters, and Boolean values, are collectively known as the *primitive data types*. The following table lists the most commonly used primitive data types and the range of values that you can store in each:

Data type	Description	Size (bits)	Range	Sample usage
int	Whole numbers (integers)	32	$-2^{31}$ through $2^{31} - 1$	int count; count = 42;
long	Whole numbers (bigger range)	64	$-2^{63}$ through $2^{63} - 1$	long wait; wait = 42L;
float	Floating-point numbers	32	$-3.4 \times 10^{38}$ through $3.4 \times 10^{38}$	float away; away = 0.42F;
double	Double-precision (more accurate) floating-point numbers	64	$\pm 5.0 \times 10^{324}$ through $\pm 1.7 \times 10^{308}$	double trouble; trouble = 0.42;
decimal	Monetary values	128	28 significant figures	decimal coin; coin = 0.42M;
string	Sequence of characters	16 bits per character	Not applicable	string vest; vest = "forty two";
char	Single character	16	A single character	char grill; grill = 'x';
bool	Boolean	8	true or false	bool teeth; teeth = false;

## Unassigned local variables

When you declare a variable, it contains a random value until you assign a value to it. This behavior was a rich source of bugs in C and C++ programs that created a variable and accidentally used it as a source of information before giving it a value.

C# does not allow you to use an unassigned variable. You must assign a value to a variable before you can use it; otherwise, your program will not compile. This requirement is called the *definite assignment rule*. For example, the following statements generate the compile-time error message, “Use of unassigned local variable ‘age’” because the `Console.WriteLine` statement attempts to display the value of an uninitialized variable:

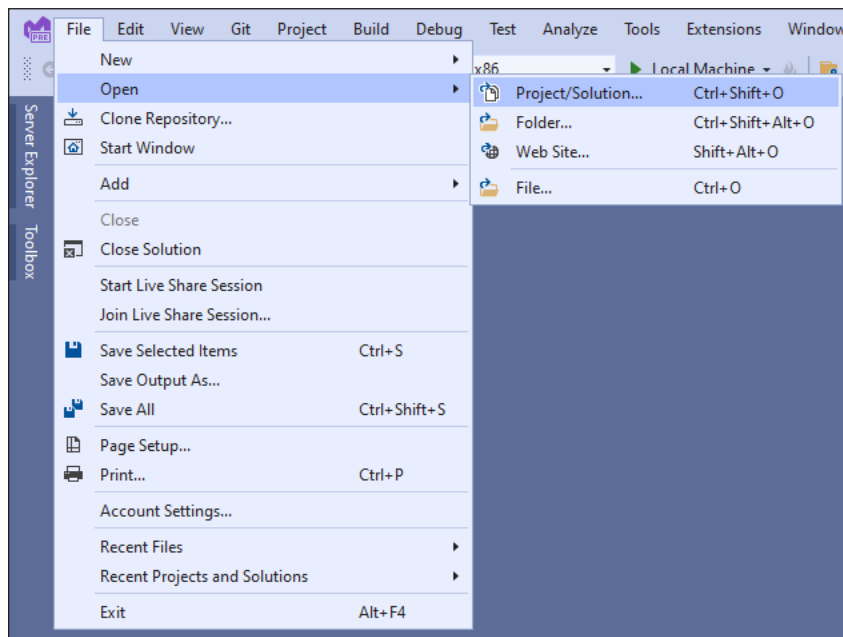
```
int age;
Console.WriteLine(age); // compile-time error
```

## Displaying primitive data type values

In the following exercise, you use a C# program named `PrimitiveDataTypes` to demonstrate how several primitive data types work.

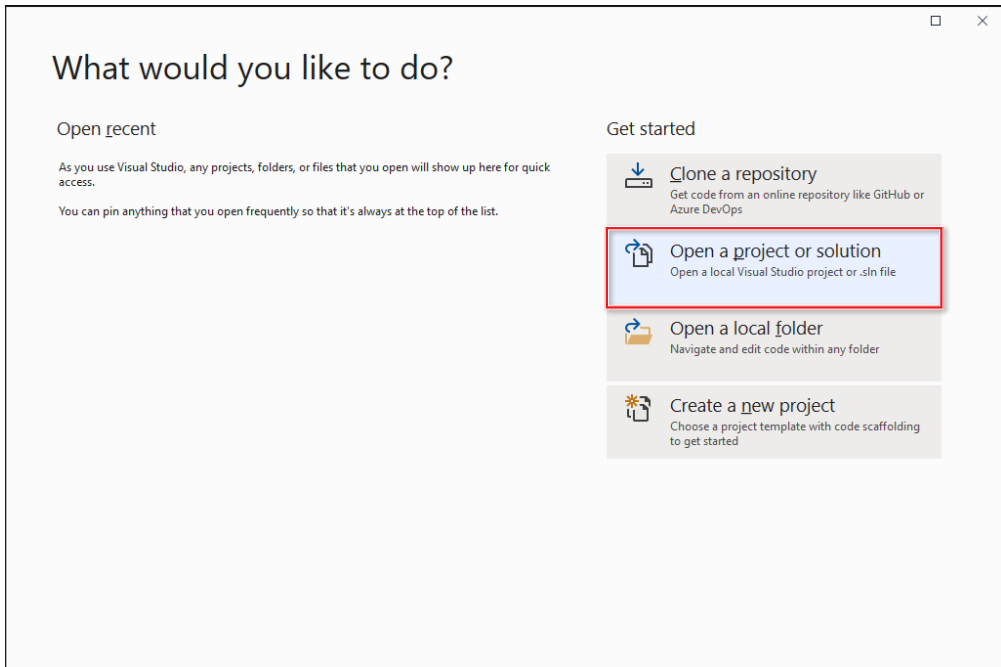
### To display primitive data type values

1. If Visual Studio 2022 is already running, on the **File** menu, select **Open**, and then select **Project/Solution**.





If Visual Studio 2022 isn't running, start it, and then select **Open a project or solution**.



The Open Project dialog opens.

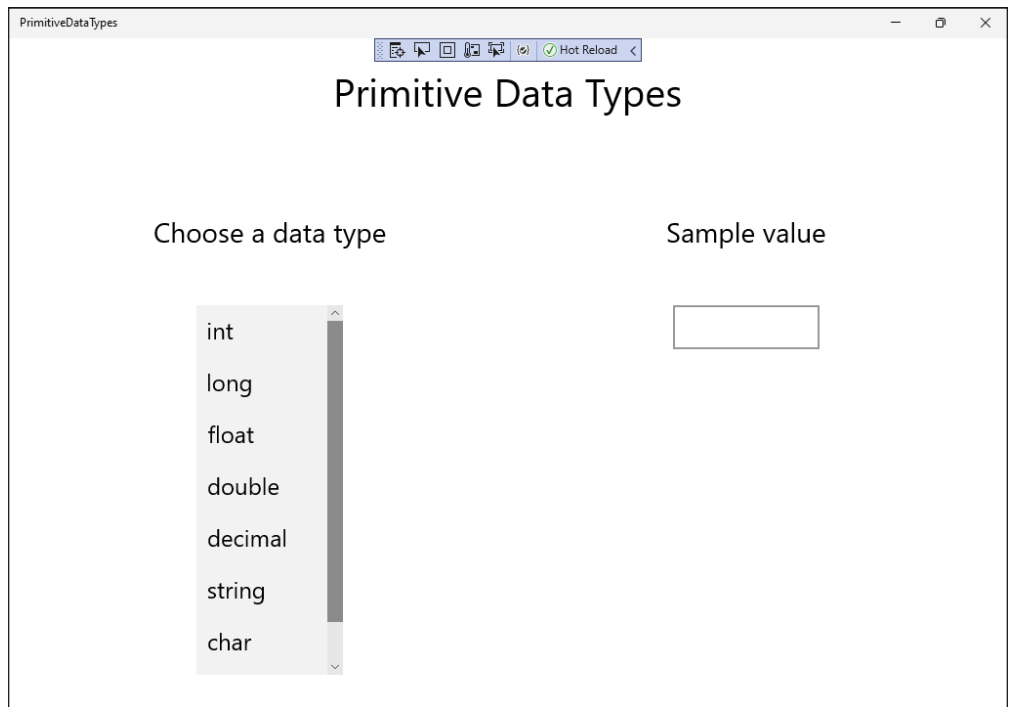
2. Navigate to the **\Microsoft Press\VCSBS\Chapter 2\PrimitiveDataTypes** folder in your **Documents** folder.
3. Select the **PrimitiveDataTypes** solution file, and then select **Open**.

The solution loads, and Solution Explorer displays the PrimitiveDataTypes project.



**Note** Solution file names have the .sln suffix, such as PrimitiveDataTypes.sln. A solution can contain one or more projects. Visual C# project files have the .csproj suffix. If you open a project rather than a solution, Visual Studio 2022 automatically creates a new solution file for it. This situation can be confusing if you're not aware of this feature because it can result in you accidentally generating multiple solutions for the same project.

4. On the **Debug** menu, choose **Start Debugging**. This enables the application to catch and report errors more easily than if you simply run the application.



5. In the **Choose a data type** list, select **string**.  
The text *forty-two* appears in the Sample value box.
6. In the **Choose a data type** list, select **int**.  
The text *to do* appears in the Sample value box, indicating that the statements to display an `int` value still need to be written.
7. Select each data type in the list. Confirm that the code for the `double` and `bool` types is not yet implemented; the application displays the results as *to do*.
8. Return to Visual Studio 2022 and then, on the **Debug** menu, select **Stop Debugging**.  
Alternatively, close the window to stop debugging.

### To use primitive data types in code

1. In Solution Explorer, expand the **PrimitiveDataTypes** project (if it is not already expanded), and then double-click **MainPage.xaml**.

The application form appears in the Design View window.



**Tip** If your screen is not big enough to display the entire form, you can zoom in and out in the Design View window by pressing Ctrl+Alt+= and Ctrl+Alt+- or by selecting the size from the Zoom dropdown list in the lower-left corner of the Design View window.

2. In the XAML pane, scroll down to locate the markup for the `ListBox` control. This control displays the list of data types in the left part of the form. It looks like this (some of the properties have been removed from this text to aid readability):

```
<ListBox x:Name="type" ... SelectionChanged="typeSelectionChanged">
 <ListBoxItem>int</ListBoxItem>
 <ListBoxItem>long</ListBoxItem>
 <ListBoxItem>float</ListBoxItem>
 <ListBoxItem>double</ListBoxItem>
 <ListBoxItem>decimal</ListBoxItem>
 <ListBoxItem>string</ListBoxItem>
 <ListBoxItem>char</ListBoxItem>
 <ListBoxItem>bool</ListBoxItem>
</ListBox>
```

The `ListBox` control displays each data type as a separate `ListBoxItem`. When the application is running, if a user selects an item in the list, the `SelectionChanged` event occurs (this is a little bit like the `Click` event that occurs when the user selects a button, as demonstrated in Chapter 1). In this case, `ListBox` invokes the `typeSelectionChanged` method. This method is defined in the `MainPage.xaml.cs` file.

3. On the **View** menu, select **Code**.

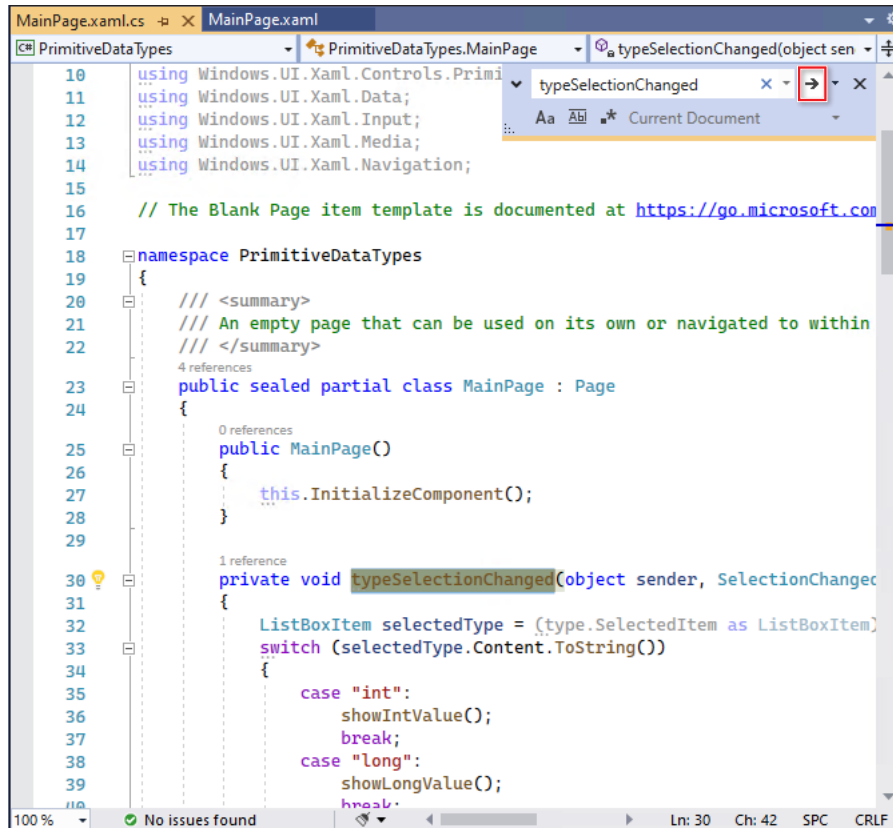
The Code and Text Editor window opens, displaying the `MainPage.xaml.cs` file.



**Note** You can also use Solution Explorer to access the code. Select the arrow to the left of the `MainPage.xaml` file to expand the node, and then double-click `MainPage.xaml.cs`.

4. In the Code and Text Editor window, locate the `typeSelectionChanged` method.

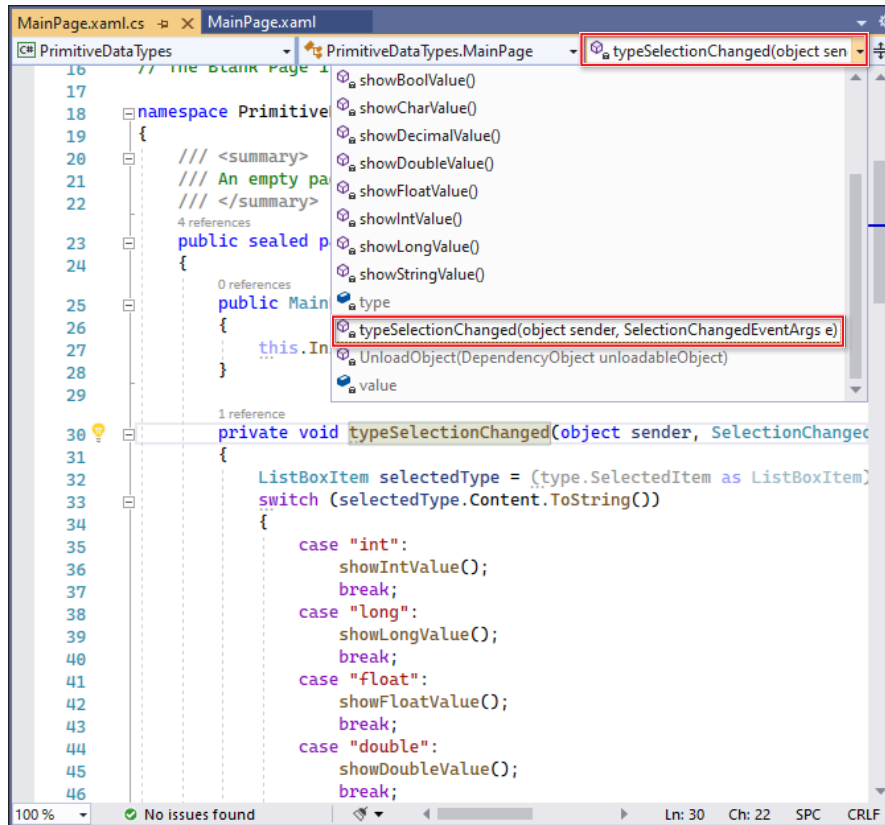
To locate an item in your project, open the **Edit** menu, select **Find And Replace**, and then select **Quick Find**. A shortcut menu opens in the upper-right corner of the Code and Text Editor window. In the text box on this shortcut menu, type the name of the item you're looking for, and then select **Find Next** (the right-arrow symbol next to the text box).



By default, the search is not case sensitive. If you want to perform a case-sensitive search, select the **Aa (Match Case)** button below the text you're searching for.

Instead of using the Edit menu, you can press **Ctrl+F** to display the Quick Find dialog. And you can press **Ctrl+H** to display the Quick Replace dialog.

As an alternative to using Quick Find, you can locate the methods in a class by using the **Class Members** dropdown list above the Code and Text Editor window, on the right. This dropdown list displays all the methods in the class, along with the variables and other items that the class contains. (You'll learn more about these items in later chapters.) In the dropdown list, choose the **typeSelectionChanged** method; the cursor moves directly to the `typeSelectionChanged` method in the class.



If you have programmed using another language, you can probably guess how the `typeSelectionChanged` method works. If not, Chapter 4, “Using decision statements,” makes this code clear. At present, all you need to understand is that when the user selects an item in the `ListBox` control, the details of the item are passed to this method, which then uses this information to determine what happens next. For example, if the user selects the float value, this method calls another method named `showFloatValue`.

5. Scroll down through the code and find the `showFloatValue` method, which looks like this:

```
private void showFloatValue()
{
 float floatVar;
 floatVar = 0.42F;
 value.Text = floatVar.ToString();
}
```

The body of this method contains three statements.

- The first statement declares a variable named `floatVar` of type `float`.
- The second statement assigns `floatVar` the value `0.42F`.



**Important** Remember that the `F` type suffix specifies that `0.42` should be treated as a float value. If you forget the `F`, the value `0.42` is treated as a double, and your program will not compile because you cannot assign a value of one type to a variable of a different type without writing additional code. `C#` is very strict in this respect.

- The third statement displays the value of this variable in the value text box on the form. This statement requires your attention. As illustrated in Chapter 1, the way you display an item in a text box is to set its `Text` property (you did this by using XAML in Chapter 1). You can also perform this task programmatically, which is what's happening here. Notice that you access the property of an object by using the same dot notation that you saw for running a method. (Remember `Console.WriteLine` from Chapter 1?) Also, the data you put in the `Text` property must be a string and not a number. If you try to assign a number to the `Text` property, your program will not compile. Fortunately, the .NET libraries provide some help in the form of the `ToString` method.

Every data type in the .NET libraries has a `ToString` method. The purpose of `ToString` is to convert an object to its string representation. The `showFloatValue` method uses the `ToString` method of the float variable `floatVar` object to generate a string version of the value of this variable. You can then safely assign this string to the `Text` property of the value text box. When you create your own data types and classes, you can define your own implementation of the `ToString` method to specify how your class should be represented as a string. You learn more about creating your own classes in Chapter 7, "Creating and managing classes and objects."

6. In the Code and Text Editor window, locate the `showIntValue` method:

```
private void showIntValue()
{
 value.Text = "to do";
}
```

The `showIntValue` method is called when you select the `int` type in the list box.

7. At the start of the `showIntValue` method, on a new line after the opening brace, add the following two statements shown in bold:

```
private void showIntValue()
{
 int intVar;
 intVar = 42;
 value.Text = "to do";
}
```

The first statement creates a variable called `intVar` that can hold an `int` value. The second statement assigns the value `42` to this variable.

8. In the original statement in this method, change the string "to do" to `intVar.ToString();`.

The method should now look exactly like this:

```
private void showIntValue()
{
 int intVar;
 intVar = 42;
 value.Text = intVar.ToString();
}
```

9. On the **Debug** menu, select **Start Debugging**.

The form appears again.

10. In the **Choose a data type** list, select **int**. Confirm that the value 42 is displayed in the Sample Value box.

11. Return to Visual Studio and then, on the **Debug** menu, select **Stop Debugging**.

12. In the Code and Text Editor window, find the `showDoubleValue` method.

13. Edit the `showDoubleValue` method exactly as shown in bold type in the following code:

```
private void showDoubleValue()
{
 double doubleVar;
 doubleVar = 0.42;
 value.Text = doubleVar.ToString();
}
```

This code is similar to the `showIntValue` method, except that it creates a variable called `doubleVar` that holds double values and is assigned the value 0.42.

14. In the Code and Text Editor window, locate the `showBoolValue` method.

15. Edit the `showBoolValue` method exactly as follows:

```
private void showBoolValue()
{
 bool boolVar;
 boolVar = false;
 value.Text = boolVar.ToString();
}
```

Again, this code is similar to the previous examples, except that `boolVar` can only hold a Boolean value, `true` or `false`. In this case, the value assigned is `false`.

16. On the **Debug** menu, select **Start Debugging**.

17. In the **Choose a data type** list, select **float**, **double**, and **bool**. In each case, verify that the correct value is displayed in the Sample Value box.

18. Return to Visual Studio and then, on the **Debug** menu, select **Stop Debugging**.

# Using arithmetic operators

C# supports the regular arithmetic operations you learned in your childhood: the plus sign (+) for addition, the minus sign (−) for subtraction, the asterisk (\*) for multiplication, and the forward slash (/) for division. The symbols +, −, \*, and / are called *operators* because they “operate” on values to create new values. In the following example, the variable `moneyPaidToConsultant` ends up holding the product of 750 (the daily rate) and 20 (the number of days the consultant was employed):

```
long moneyPaidToConsultant;
moneyPaidToConsultant = 750 * 20;
```



**Note** The values on which an operator performs its function are called *operands*. In the expression `750 * 20`, the `*` is the operator, and 750 and 20 are the operands.

## Operators and types

Not all operators apply to all data types. The operators that you can use on a value depend on the value’s type. For example, you can use all the arithmetic operators on values of type `char`, `int`, `long`, `float`, `double`, or `decimal`. However, except for the plus operator (+), you can’t use the arithmetic operators on values of type `string`, and you cannot use any of them with values of type `bool`. So, the following statement is not allowed because the `string` type doesn’t support the minus operator (subtracting one string from another is meaningless):

```
// compile-time error
Console.WriteLine("Gillingham" - "Forest Green Rovers");
```

However, you can use the + operator to concatenate string values. You need to be careful because this can have unexpected results. For example, the following statement writes 431 (not 44) to the console:

```
Console.WriteLine("43" + "1");
```



**Tip** The .NET libraries provide a method called `Int32.Parse` that you can use to convert a string value to an integer if you need to perform arithmetic computations on values held as strings.



## String interpolation

C# renders many uses of the + operator obsolete for concatenating strings.

A common use of string concatenation is to generate string values that include variable values. For example, consider the following statements:

```
string username = "John";
string message = "Hello " + username;
```

String interpolation lets you use the following syntax instead:

```
string username = "John";
string message = ($"Hello {username }");
```

The \$ symbol at the start of the string indicates that it is an interpolated string and that any expressions between the { and } characters should be evaluated, and the result substituted in their place. Without the leading \$ symbol, the string {username} would be treated literally.

String interpolation is more efficient than using the + operator; string concatenation using the + operator can be memory hungry because of how strings are handled by the .NET libraries. String interpolation is also arguably more readable and less error prone.

You should also be aware that the type of the result of an arithmetic operation depends on the type of the operands used. For example, the value of the expression `5.0/2.0` is `2.5`; the type of both operands is `double`, so the type of the result is also `double`. (Remember that in C#, literal numbers with decimal points are always `double`, not `float`, to maintain as much accuracy as possible.) However, the value of the expression `5/2` is `2`. In this case, the type of both operands is `int`, so the type of the result is also `int`. C# always rounds toward zero in circumstances like this.

The situation gets a little more complicated if you mix the types of the operands. For example, the expression `5/2.0` consists of an `int` and a `double`. The C# compiler detects the mismatch and generates code that converts the `int` into a `double` before performing the operation. The result of the operation is, therefore, a `double` (`2.5`). However, although this works, it is considered poor practice to mix types in this way.

C# also supports a less-familiar arithmetic operator: the *remainder*, or *modulus*, operator, which is represented by the percent sign (%). The result of `x % y` is the integer remainder after dividing the integer value `x` by the integer value `y`. So, for example, `9 % 2` is `1` because 9 divided by 2 is 4, with a remainder of 1.



**Note** If you are familiar with C or C++, you know that you can't use the remainder operator on `float` or `double` values in these languages. However, C# relaxes this rule. The remainder operator is valid with all numeric types, and the result is not necessarily an integer. For example, the result of the expression `7.0 % 2.4` is `2.2`.

## Numeric types and infinite values

There are other features of numbers in C# that you should be aware of. For example, the result of dividing any number by zero is infinity, which is outside the range of the `int`, `long`, and `decimal` types; consequently, evaluating an expression such as `5/0` results in error. However, the `double` and `float` types have a special value that can represent infinity, and the value of the expression `5.0/0.0` is `Infinity`. The one exception to this rule is the value of the expression `0.0/0.0`. Usually, if you divide zero by anything, the result is zero, but if you divide anything by zero, the result is infinity. The expression `0.0/0.0` results in a paradox; the value must be zero and infinity at the same time. C# has another special value for this situation, called `NaN`, which stands for *not a number*. So if you evaluate `0.0/0.0`, the result is `NaN`.

`NaN` and `Infinity` propagate through expressions. If you evaluate `10 + NaN`, the result is `NaN`, and if you evaluate `10 + Infinity`, the result is `Infinity`. The value of the expression `Infinity * 0` is `NaN`.

## Examining arithmetic operators

The following exercise demonstrates how to use the arithmetic operators on `int` values.

### To run the MathOperators project

1. Start Visual Studio 2022 if it is not already running.
2. Open the **MathOperators** solution, located in the `\Microsoft Press\VCSBS\Chapter 2\MathOperators` folder in your **Documents** folder.
3. On the **Debug** menu, select **Start Debugging**.

The following form appears:

MathOperators

Left Operand

Right Operand

☐ + Addition

☐ - Subtraction

☐ \* Multiplication

☐ / Division

☐ % Remainder

Calculate

Expression:

Result:

4. In the **Left Operand** box, type **54**.
5. In the **Right Operand** box, type **13**.

You can now apply any of the operators to the values in the text boxes.

6. Select the **– Subtraction** option, and then select **Calculate**.

The text in the Expression box changes to  $54 - 13$ , but the value 0 appears in the Result box. This is clearly wrong.

7. Select the **/ Division** option, and then select **Calculate**.

The text in the Expression box changes to  $54 / 13$ , and again the value 0 appears in the Result box.

8. Select the **% Remainder** option, and then select **Calculate**.

The text in the Expression box changes to  $54 \% 13$ , but, once again, the value 0 appears in the Result text box. Test other combinations of numbers and operators; you'll find that they all currently yield the value 0.



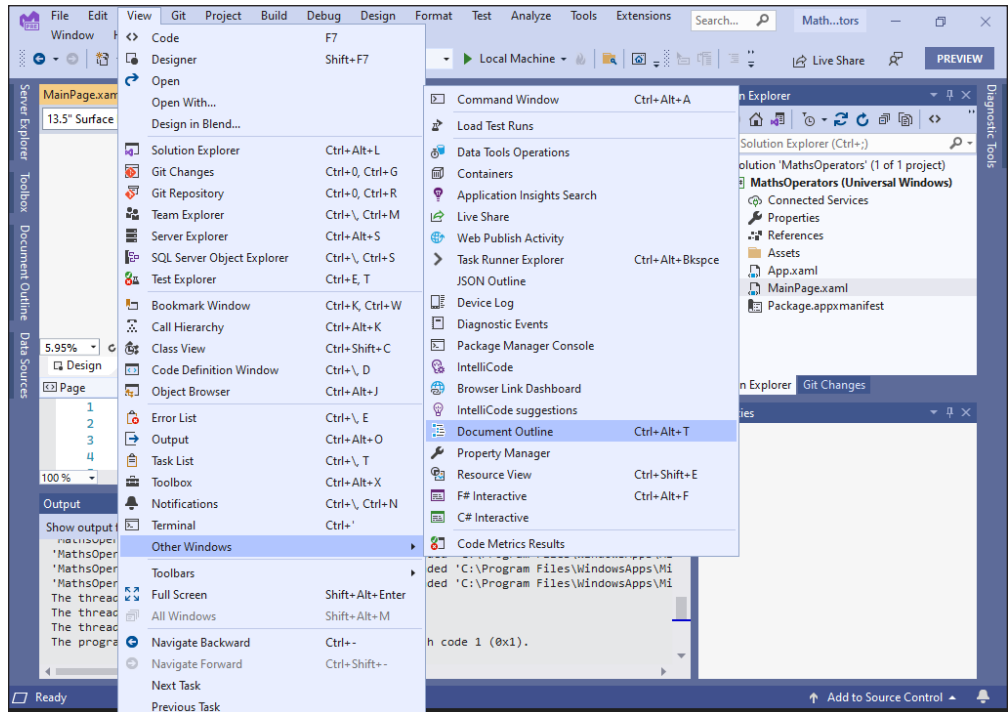
**Note** If you type a non-integer value into either of the operand boxes, the application detects an error and displays the message "Input string was not in a correct format." You'll learn more about how to catch and handle errors and exceptions in Chapter 6, "Managing errors and exceptions."

9. When you're finished, return to Visual Studio and then, on the **Debug** menu, select **Stop Debugging**.

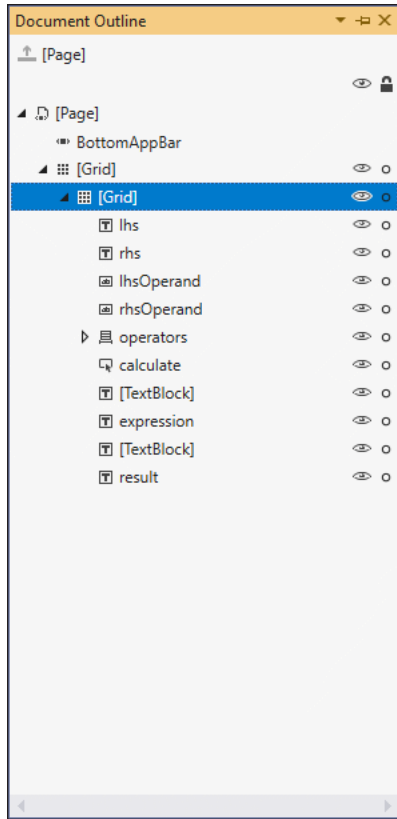
As you might have guessed, none of the calculations is currently implemented by the MathOperators application. In the next exercise, you'll correct this.

## To perform calculations in the MathOperators application

1. Display the MainPage.xaml form in the Design View window. (In Solution Explorer, in the **MathOperators** project, open the file **MainPage.xaml**.)
2. On the **View** menu, point to **Other Windows**, and then select **Document Outline**.



The Document Outline window appears, showing the names and types of the controls on the form. The Document Outline window provides a simple way to locate and select controls on a complex form. The controls are arranged in a hierarchy, starting with the Page that constitutes the form. As mentioned in Chapter 1, a Universal Windows Platform (UWP) app page contains a Grid control, and the other controls are placed within this Grid. If you expand the Grid node in the Document Outline window, the other controls appear, starting with another Grid (the outer Grid acts as a frame, and the inner Grid contains the controls that you see on the form). If you expand the inner Grid, you can see each of the controls on the form.



If you select any of these controls, the corresponding element is highlighted in the Design View window. Similarly, if you select a control in the Design View window, the corresponding control is selected in the Document Outline window. (To see this in action, pin the Document Outline window in place by deselecting the **Auto Hide** button in the upper-right corner of the Document Outline window.)

3. On the form, select the two TextBox controls into which the user types numbers. In the Document Outline window, verify that they are named `lhsOperand` and `rhsOperand`.

When the form runs, the `Text` property of each of these controls holds the values that the user enters.

4. Toward the bottom of the form, verify that the `TextBlock` control used to display the expression being evaluated is named `expression` and that the `TextBlock` control used to display the result of the calculation is named `result`.
5. Close the Document Outline window.
6. On the **View** menu, select **Code** to display the code for the `MainPage.xaml.cs` file in the Code and Text Editor window.

7. In the Code and Text Editor window, locate the `addValues` method. It looks like this:

```
private void addValues()
{
 int lhs = int.Parse(lhsOperand.Text);
 int rhs = int.Parse(rhsOperand.Text);
 int outcome = 0;
 // TODO: Add rhs to lhs and store the result in outcome
 expression.Text = $"{lhsOperand.Text} + {rhsOperand.Text}";
 result.Text = outcome.ToString();
}
```

The body of this method contains six statements:

- The first statement in this method declares an `int` variable called `lhs` and initializes it with the integer corresponding to the value typed by the user in the `lhsOperand` box. Remember that the `Text` property of a `TextBox` control contains a string, but `lhs` is an `int`, so you must convert this string to an integer before you can assign it to `lhs`. The `int` data type provides the `int.Parse` method, which does precisely this.
  - The second statement declares an `int` variable called `rhs` and initializes it to the value in the `rhsOperand` box after converting it to an `int`.
  - The third statement declares an `int` variable called `outcome`.
  - The fourth statement consists of a comment that states that you need to add `rhs` to `lhs` and store the result in `outcome`. This is the missing bit of code that you need to implement, which you'll do in the next step.
  - The fifth statement uses string interpolation to construct a string that indicates the calculation being performed and assigns the result to the `expression.Text` property. This causes the string to appear in the `Expression` box on the form.
  - The sixth statement displays the result of the calculation by assigning it to the `Text` property of the `Result` box. Remember that the `Text` property is a string, and the result of the calculation is an `int`, so you must convert the `int` to a string before assigning it to the `Text` property. Recall that this is what the `ToString` method of the `int` type does.
8. Below the comment in the middle of the `addValues` method, add the following statement (shown here in bold):

```
private void addValues()
{
 int lhs = int.Parse(lhsOperand.Text);
 int rhs = int.Parse(rhsOperand.Text);
 int outcome = 0;
 // TODO: Add rhs to lhs and store the result in outcome
 outcome = lhs + rhs;
 expression.Text = $"{lhsOperand.Text} + {rhsOperand.Text}";
 result.Text = outcome.ToString();
}
```

This statement evaluates the expression `lhs + rhs` and stores the result in `outcome`.

9. Examine the `subtractValues` method. You should see that it follows a similar pattern. Here you need to add the statement to calculate the result of subtracting `rhs` from `lhs` and store it in `outcome`. Add the following statement (in bold) to this method:

```
private void subtractValues()
{
 int lhs = int.Parse(lhsOperand.Text);
 int rhs = int.Parse(rhsOperand.Text);
 int outcome = 0;
 // TODO: Subtract rhs from lhs and store the result in outcome
 outcome = lhs - rhs;
 expression.Text = $"{lhsOperand.Text} - {rhsOperand.Text}";
 result.Text = outcome.ToString();
}
```

10. Examine the `multiplyValues`, `divideValues`, and `remainderValues` methods. Again, they are all missing the crucial statement that performs the specified calculation. Add the appropriate statements to these methods (shown in bold):

```
private void multiplyValues()
{
 int lhs = int.Parse(lhsOperand.Text);
 int rhs = int.Parse(rhsOperand.Text);
 int outcome = 0;
 // TODO: Multiply lhs by rhs and store the result in outcome
 outcome = lhs * rhs;
 expression.Text = $"{lhsOperand.Text} * {rhsOperand.Text}";
 result.Text = outcome.ToString();
}

private void divideValues()
{
 int lhs = int.Parse(lhsOperand.Text);
 int rhs = int.Parse(rhsOperand.Text);
 int outcome = 0;
 // TODO: Divide lhs by rhs and store the result in outcome
 outcome = lhs / rhs;
 expression.Text = $"{lhsOperand.Text} / {rhsOperand.Text}";
 result.Text = outcome.ToString();
}

private void remainderValues()
{
 int lhs = int.Parse(lhsOperand.Text);
 int rhs = int.Parse(rhsOperand.Text);
 int outcome = 0;
 // TODO: Calculate the remainder after dividing lhs by rhs and store the result in outcome
 outcome = lhs % rhs;
 expression.Text = $"{lhsOperand.Text} % {rhsOperand.Text}";
 result.Text = outcome.ToString();
}
```

## To test the MathsOperators application

1. On the **Debug** menu, select **Start Debugging** to build and run the application.
2. Type **54** in the **Left Operand** box, type **13** in the **Right Operand** box, select the **+ Addition** option, and then select **Calculate**.  
  
The value 67 should appear in the Result box.
3. Select the **- Subtraction** option, and then select **Calculate**. Verify that the result is now 41.
4. Select the **\* Multiplication** option, and then select **Calculate**. Verify that the result is now 702.
5. Select the **/ Division** option, and then select **Calculate**. Verify that the result is now 4.  
  
In real life, 54/13 is 4.153846 recurring, but this is not real life; this is C# performing integer division. When you divide one integer by another integer, the answer you get back is an integer, as explained earlier.
6. Select the **% Remainder** option, and then select **Calculate**. Verify that the result is now 2.  
  
When dealing with integers, the remainder after dividing 54 by 13 is 2;  $(54 - ((54/13) * 13))$  is 2. This is because the calculation rounds down to an integer at each stage. (My grammar school mathematics teacher would be horrified to be told that  $(54/13) * 13$  does not equal 54!)
7. Return to Visual Studio and stop debugging.

## Controlling precedence

*Precedence* governs the order in which an expression's operators are evaluated. Consider the following expression, which uses the + and \* operators:

$2 + 3 * 4$

This expression is potentially ambiguous: Do you perform the addition first or the multiplication? The order of the operations matters because it changes the result:

- If you perform the addition first, followed by the multiplication, the result of the addition  $(2 + 3)$  forms the left operand of the \* operator, and the result of the whole expression is  $5 * 4$ , which is 20.
- If you perform the multiplication first, followed by the addition, the result of the multiplication  $(3 * 4)$  forms the right operand of the + operator, and the result of the whole expression is  $2 + 12$ , which is 14.

In C#, the multiplicative operators (\*, /, and %) have precedence over the additive operators (+ and -), so in expressions such as  $2 + 3 * 4$ , the multiplication is performed first, followed by the addition. The answer to  $2 + 3 * 4$  is therefore 14.



You can use parentheses to override precedence and force operands to bind to operators in a different way. For example, in the following expression, the parentheses force the 2 and the 3 to bind to the + operator (making 5), and the result of this addition forms the left operand of the \* operator to produce the value 20:

`(2 + 3) * 4`



**Note** The term *parentheses* or *round brackets* refers to ( ). The term *braces* or *curly brackets* refers to { }. The term *square brackets* refers to [ ].

## Using associativity to evaluate expressions

Operator precedence is only half the story. What happens when an expression contains different operators that have the same precedence? This is where *associativity* becomes important. *Associativity* is the direction (left or right) in which the operands of an operator are evaluated. Consider the following expression that uses the / and \* operators:

`4 / 2 * 6`

At first glance, this expression is potentially ambiguous. Do you perform the division first or the multiplication? The precedence of both operators is the same (they are both multiplicative operators), but the order in which the operators in the expression are applied is important because you can get two different results:

- If you perform the division first, the result of the division (`4 / 2`) forms the left operand of the \* operator, and the result of the whole expression is `(4 / 2) * 6`, or 12.
- If you perform the multiplication first, the result of the multiplication (`2 * 6`) forms the right operand of the / operator, and the result of the whole expression is `4 / (2 * 6)`, or 4 / 12.

In this case, the associativity of the operators determines how the expression is evaluated. The \* and / operators are both left associative, which means that the operands are evaluated from left to right. In this case, `4 / 2` will be evaluated before multiplying by 6, giving the result 12.

## Associativity and the assignment operator

In C#, the equal sign (=) is an operator. All operators return a value based on their operands. The assignment operator = is no different. It takes two operands: the operand on the right side is evaluated and then stored in the operand on the left side. The value of the assignment operator is the value that was assigned to the left operand. For example, in the following assignment statement, the value returned by the assignment operator is 10, which is also the value assigned to the variable `myInt`:

```
int myInt;
myInt = 10; // value of assignment expression is 10
```

At this point, you might be thinking that this is all very nice and esoteric, but so what? Well, because the assignment operator returns a value, you can use this same value with another occurrence of the assignment statement, like this:

```
int myInt;
int myInt2;
myInt2 = myInt = 10;
```

The value assigned to the variable `myInt2` is the value that was assigned to `myInt`. The assignment statement assigns the same value to both variables. This technique is useful if you want to initialize several variables to the same value. It makes it very clear to anyone reading your code that all the variables must have the same value:

```
myInt5 = myInt4 = myInt3 = myInt2 = myInt = 10;
```

From this discussion, you can probably deduce that the assignment operator associates from right to left. The rightmost assignment occurs first, and the value assigned propagates through the variables from right to left. If any of the variables previously had a value, it is overwritten by the value now being assigned.

You should treat this construct with caution, however. One frequent mistake that new C# programmers make is to try to combine this use of the assignment operator with variable declarations. For example, you might expect the following code to create and initialize three variables with the same value (10):

```
int myInt, myInt2, myInt3 = 10;
```

This is legal C# code (because it compiles). What it does is declare the variables `myInt`, `myInt2`, and `myInt3` and initialize `myInt3` with the value 10. However, it does not initialize `myInt` or `myInt2`. If you try to use `myInt` or `myInt2` in an expression such as

```
myInt3 = myInt / myInt2;
```

the compiler generates the following errors:

```
Use of unassigned local variable 'myInt'
Use of unassigned local variable 'myInt2'
```

## Incrementing and decrementing variables

---

If you want to add 1 to a variable, you can use the `+` operator, as demonstrated here:

```
count = count + 1;
```

However, adding 1 to a variable is so common that C# provides its own operator just for this purpose: the `++` operator. To increment the variable `count` by 1, you can write the following statement:

```
count++;
```

Similarly, C# provides the `-` operator, which you can use to subtract 1 from a variable, like this:

```
count--;
```

The `++` and `--` operators are *unary* operators, meaning they take only a single operand. They share the same precedence and are both left associative.

## Prefix and postfix

The increment (++) and decrement (--) operators are unusual in that you can place them either before or after the variable. Placing the operator symbol before the variable is called the *prefix form* of the operator, and using the operator symbol after the variable is called the *postfix form*. Here are examples:

```
count++; // postfix increment
++count; // prefix increment
count--; // postfix decrement
--count; // prefix decrement
```

Whether you use the prefix or postfix form of the ++ or -- operator makes no difference to the variable being incremented or decremented. For example, if you write `count++`, the value of `count` increases by 1, and if you write `++count`, the value of `count` also increases by 1. Knowing this, you're probably wondering why there are two ways to write the same thing. To understand the answer, you must remember that ++ and -- are operators, and that all operators are used to evaluate an expression that has a value. The value returned by `count++` is the value of `count` before the increment takes place, whereas the value returned by `++count` is the value of `count` after the increment takes place. Here's an example:

```
int x;
x = 42;
Console.WriteLine(x++); // x is now 43, 42 written out
x = 42;
Console.WriteLine(++x); // x is now 43, 43 written out
```

The way to remember which operand does what is to look at the order of the elements (the operand and the operator) in a prefix or postfix expression. In the expression `x++`, the variable `x` occurs first, so its value is used as the value of the expression before `x` is incremented. In the expression `++x`, the operator occurs first, so its operation is performed before the value of `x` is evaluated as the result.

These operators are most commonly used in `while` and `do` statements, which are presented in Chapter 5, "Using compound assignment and iteration statements." If you're using the increment and decrement operators in isolation, stick to the postfix form and be consistent.

## Declaring implicitly typed local variables

---

Earlier in this chapter, you saw that you declare a variable by specifying a data type and an identifier, like this:

```
int myInt;
```

It was also mentioned that you should assign a value to a variable before you attempt to use it. You can declare and initialize a variable in the same statement, such as illustrated in the following:

```
int myInt = 99;
```

Or, you can even do it like this, assuming that `myOtherInt` is an initialized integer variable:

```
int myInt = myOtherInt * 99;
```

Remember that the value you assign to a variable must be of the same type as the variable. For example, you can assign an `int` value only to an `int` variable. The C# compiler can quickly determine the type of an expression used to initialize a variable and indicate whether it matches or doesn't match the type of the variable.

You can also ask the C# compiler to infer the type of a variable from an expression and use this type when declaring the variable by using the `var` keyword in place of the type, as demonstrated here:

```
var myVariable = 99;
var myOtherVariable = "Hello";
```

The variables `myVariable` and `myOtherVariable` are referred to as *implicitly typed* variables. The `var` keyword causes the compiler to deduce the type of the variables from the types of the expressions used to initialize them. In these examples, `myVariable` is an `int`, and `myOtherVariable` is a `string`. However, it's important to understand that this is a convenience for declaring variables only. After a variable has been declared, you can assign only values of the inferred type to it. For example, you cannot assign `float`, `double`, or `string` values to `myVariable` at a later point in your program. You should also understand that you can use the `var` keyword only when you supply an expression to initialize a variable. The following declaration is illegal and causes a compilation error:

```
var yetAnotherVariable; // Error - compiler cannot infer type
```



**Important** If you have programmed with Visual Basic in the past, you might be familiar with the `Variant` type, which you can use to store any type of value in a variable. I emphasize here and now that you should forget everything you ever learned when programming with Visual Basic about `Variant` variables. Although the keywords look similar, `var` and `Variant` mean totally different things. When you declare a variable in C# by using the `var` keyword, the type of values that you assign to the variable *cannot change* from that used to initialize the variable.

If you are a purist, you're probably gritting your teeth at this point and wondering why on Earth the designers of a neat language such as C# should allow a feature such as `var` to creep in. After all, it sounds like an excuse for extreme laziness on the part of programmers and can make it more difficult to understand what a program is doing or track down bugs (and it can even easily introduce new bugs into your code). However, `var` has a very valid place in C#, as you'll see when you progress through many of the following chapters. For the time being, though, we'll stick to using explicitly typed variables except for when implicit typing becomes a necessity.

## Summary

In this chapter, you saw how to create and use variables and learned about some of the common data types available for variables in C#. You also learned about identifiers. In addition, you used several operators to build expressions, and you learned how the precedence and associativity of operators determine how expressions are evaluated.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 3.
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

## Quick Reference

To	Do this
Declare a variable	Write the name of the data type, followed by the name of the variable, followed by a semicolon. For example: <code>int outcome;</code>
Declare a variable and give it an initial value	Write the name of the data type, followed by the name of the variable, followed by the assignment operator and the initial value. Finish with a semicolon. For example: <code>int outcome = 99;</code>
Change the value of a variable	Write the name of the variable on the left, followed by the assignment operator, followed by the expression calculating the new value, followed by a semicolon. For example: <code>outcome = 42;</code>
Generate a string representation of the value in a variable	Call the <code>ToString</code> method of the variable. For example: <code>int intVar = 42; string stringVar = intVar.ToString();</code>
Convert a string to an int	Call the <code>System.Int32.Parse</code> method. For example: <code>string stringVar = "42"; int intVar = System.Int32.Parse(stringVar);</code>
Override the precedence of an operator	Use parentheses in the expression to force the order of evaluation. For example: <code>(3 + 4) * 5</code>
Assign the same value to several variables	Use an assignment statement that lists all the variables. For example: <code>myInt4 = myInt3 = myInt2 = myInt = 10;</code>
Increment or decrement a variable	Use the <code>++</code> or <code>--</code> operator. For example: <code>count++;</code>

# Writing methods and applying scope

**After completing this chapter, you will be able to:**

- Declare methods, pass information to a method, and return information from a method.
- Call methods.
- Define local and class scope.
- Use the integrated debugger to step into and out of methods as they run.
- Use optional parameters and named arguments with methods.

In Chapter 2, “Working with variables, operators, and expressions,” you learned how to declare variables, how to create expressions using operators, and how precedence and associativity control the way in which expressions containing multiple operators are evaluated. In this chapter, you’ll learn about methods. You’ll see how to declare and call methods, how to use arguments and parameters to pass information to a method, and how to return information from a method by using a `return` statement. You’ll also see how to step into and out of methods by using the Microsoft Visual Studio 2022 integrated debugger. This information is useful when you need to trace the execution of your methods because they do not work quite as you expect. Finally, you’ll learn how to declare methods that take optional parameters and how to invoke methods by using named arguments.

## Creating methods

---

A *method* is a named sequence of statements. If you have programmed previously using a language such as C, C++, or Microsoft Visual Basic, you’ll see that a method is similar to a function or a subroutine.

A method has a name and a body. The method name should be a meaningful identifier that indicates the overall purpose of the method (`calculateIncomeTax`, for example). The method body contains the actual statements to be run when the method is called. Additionally, methods can be given some data for processing and can return information, which is usually the result of the processing. Methods are a fundamental and powerful mechanism.

## Declaring a method

The syntax for declaring a C# method is as follows:

```
returnType methodName(parameterList)
{
 // method body statements go here
}
```

The following is a description of the elements that make up a declaration:

- The *returnType* is the name of a type and specifies the kind of information the method returns as a result of its processing. This can be any type, such as `int` or `string`. If you're writing a method that does not return a value, you must use the keyword `void` in place of the return type.
- The *methodName* is the name used to call the method. Method names follow the same identifier rules as variable names. For example, `addValues` is a valid method name, whereas `add$Values` is not. For now, you should follow the camelCase convention for method names—for example, `displayCustomer`.
- The *parameterList* is optional and describes the types and names of the information that you can pass into a method for it to process. You write the parameters between opening and closing parentheses, `( )`, as though you're declaring variables, with the name of the type followed by the name of the parameter. If the method you're writing has two or more parameters, you must separate them with commas.
- The method body statements are the lines of code that run when the method is called. They are enclosed between opening and closing braces, `{ }`.



**Important** If you program in C, C++, or Visual Basic, you should note that C# does not support global methods. You must write all your methods inside a class; otherwise, your code will not compile.

Here's the definition of a method called `addValues` that returns an `int` result and has two `int` parameters, `leftHandSide` and `rightHandSide`:

```
int addValues(int leftHandSide, int rightHandSide)
{
 // ...
 // method body statements go here
 // ...
}
```



**Note** You must explicitly specify the types of any parameters and the return type of a method. You cannot use the `var` keyword.

Here's the definition of a method called `showResult` that does not return a value and has a single `int` parameter called `answer`:

```
void showResult(int answer)
{
 // ...
}
```

Notice the use of the keyword `void` to indicate that the method does not return anything.



**Important** If you're familiar with Visual Basic, notice that C# does not use different keywords to distinguish between a method that returns a value (a function) and a method that does not return a value (a procedure or subroutine). You must always specify either a return type or `void`.

## Returning data from a method

If you want a method to return information (that is, its return type is not `void`), you must include a `return` statement at the end of the processing in the method body. A `return` statement consists of the keyword `return` followed by an expression that specifies the returned value and a semicolon. The type of the expression must be the same as the type specified by the method declaration. For example, if a method returns an `int`, the `return` statement must return an `int`; otherwise, your program will not compile. Here is an example of a method with a `return` statement:

```
int addValues(int leftHandSide, int rightHandSide)
{
 // ...
 return leftHandSide + rightHandSide;
}
```

The `return` statement is usually positioned at the end of the method because it causes the method to finish and returns control to the statement that called the method, as described later in this chapter. Any statements that occur after the `return` statement are not executed. (The compiler warns you about this if you place statements after the `return` statement.)

If you don't want your method to return information (that is, its return type is `void`), you can use a variation of the `return` statement to cause an immediate exit from the method. You write the keyword `return` and follow it immediately with a semicolon. For example:

```
void showResult(int answer)
{
 // display the answer
 Console.WriteLine($"The answer is {answer}");
 return;
}
```

If your method does not return anything, you can also omit the `return` statement because the method finishes automatically when execution arrives at the closing brace at the end of the method. Although this practice is common, it is not always considered good style.



## Using expression-bodied methods

Some methods can be very simple, performing a single task or returning the results of a calculation without involving any additional logic. C# supports a simplified form for methods that comprise a single expression. These methods can still take parameters and return values, and they operate in the same way as the methods that you have seen so far. The following code examples show simplified versions of the `addValues` and `showResult` methods written as expression-bodied methods:

```
int addValues(int leftHandSide, int rightHandSide) => leftHandSide + rightHandSide;
void showResult(int answer) => Console.WriteLine($"The answer is {answer}");
```

The main differences are the use of the `=>` operator to reference the expression that forms the body of the method and the absence of a return statement. The value of the expression is used as the return value; if the expression does not return a value, then the method is void.

There is actually no difference in functionality between using an ordinary method and an expression-bodied method; an expression-bodied method is merely a syntactic convenience. However, you will see examples later in the book where expression-bodied methods can clarify a program by removing lots of extraneous `{` and `}` characters, making the code easier to read.

In the following exercise, you'll examine another version of the `MathOperators` project from Chapter 2. This version has been improved by the careful use of some small methods. Dividing code in this way helps to make it easier to understand and more maintainable.

### To examine method definitions

1. Start Visual Studio 2022, if it is not already running.
2. Open the **Methods** solution, which is in the `\Microsoft Press\VCSBS\Chapter 3\Methods` folder in your **Documents** folder.
3. On the **Debug** menu, select **Start Debugging**.

Visual Studio 2022 builds and runs the application. It should look the same as the application from Chapter 2.

4. Refamiliarize yourself with the application and how it works and then return to Visual Studio.
5. On the **Debug** menu, select **Stop Debugging**.
6. Display the code for `MainPage.xaml.cs` in the Code and Text Editor window (in Solution Explorer, expand the **MainPage.xaml** file and then double-click **MainPage.xaml.cs**).
7. In the Code and Text Editor window, locate the `addValues` method, which looks like this:

```
private int addValues(int leftHandSide, int rightHandSide)
{
 expression.Text = $"{leftHandSide} + {rightHandSide}";
 return leftHandSide + rightHandSide;
}
```



**Note** For the moment, don't worry about the `private` keyword at the start of the definition of this method. You'll learn what this keyword means in Chapter 7, "Creating and managing classes and objects."

The `addValues` method contains two statements.

- The first statement displays the calculation being performed in the expression box on the form.
- The second statement uses the `int` version of the `+` operator to add the values of the `leftHandSide` and `rightHandSide` `int` variables and then returns the result of this operation. Remember that adding two `int` values together creates another `int` value, so the return type of the `addValues` method is `int`.

If you look at the `subtractValues`, `multiplyValues`, `divideValues`, and `remainderValues` methods, you will see that they follow a similar pattern.

8. In the Code and Text Editor window, locate the `showResult` method, which looks like this:

```
private void showResult(int answer) => result.Text = answer.ToString();
```

This is an expression-bodied method that displays a string representation of the `answer` parameter in the result box. It does not return a value, so the type of this method is `void`.



**Tip** There is no minimum length for a method. If a method helps to avoid repetition and makes your program easier to understand, the method is useful regardless of how small it is.

There is also no maximum length for a method, but usually you want to keep your method code small enough to get the job done. If your method is more than one screen in length, consider breaking it into smaller methods for readability.

## Calling methods

Methods exist to be called! You call a method by name to ask it to perform its task. If the method requires information (as specified by its parameters), you must supply the information requested. If the method returns information (as specified by its return type), you should arrange to capture this information somehow.

## Specifying the method call syntax

The syntax of a C# method call is as follows:

```
result = methodName(argumentList)
```

The following is a description of the elements that make up a method call:

- The *methodName* must exactly match the name of the method you're calling. Remember, C# is a case-sensitive language.
- The *result =* clause is optional. If specified, the variable identified by *result* contains the value returned by the method. If the method is *void* (that is, it does not return a value), you must omit the *result =* clause of the statement. If you don't specify the *result =* clause and the method does return a value, the method runs, but the return value is discarded.
- The *argumentList* supplies the information that the method accepts. You must supply an argument for each parameter, and the value of each argument must be compatible with the type of its corresponding parameter. If the method you're calling has two or more parameters, you must separate the arguments with commas.



**Important** You must include the parentheses in every method call, even when calling a method that has no arguments.

To clarify these points, take a look at the `addValues` method again:

```
int addValues(int leftHandSide, int rightHandSide)
{
 // ...
}
```

The `addValues` method has two `int` parameters, so you must call it with two comma-separated `int` arguments, such as this:

```
addValues(39, 3); // okay
```

You can also replace the literal values 39 and 3 with the names of `int` variables. The values in those variables are then passed to the method as its arguments, like this:

```
int arg1 = 99;
int arg2 = 1;
addValues(arg1, arg2);
```

If you try to call `addValues` in some other way, you will probably not succeed for the reasons described in the following examples:

```
addValues; // compile-time error, no parentheses
addValues(); // compile-time error, not enough arguments
addValues(39); // compile-time error, not enough arguments
addValues("39", "3"); // compile-time error, wrong types for arguments
```

The `addValues` method returns an `int` value. You can use this `int` value wherever an `int` value can be used. Consider these examples:

```
int result = addValues(39, 3); // on right-hand side of an assignment
showResult(addValues(39, 3)); // as argument to another method call
```

The following exercise continues with the `Methods` application. This time, you'll examine some method calls.

## To examine method calls

1. Return to the Methods project. (This project is already open in Visual Studio 2022 if you're continuing from the previous exercise. If you are not, open it from the **\Microsoft Press\VCSBS\Chapter 3\Methods** folder in your **Documents** folder.)
2. Display the code for `MainPage.xaml.cs` in the Code and Text Editor window.
3. Locate the `calculateClick` method. Then look at the first two statements of this method after the `try` statement and opening brace. (You'll learn about `try` statements in Chapter 6, "Managing errors and exceptions.")

These statements look like this:

```
int leftHandSide = System.Int32.Parse(lhsOperand.Text);
int rightHandSide = System.Int32.Parse(rhsOperand.Text);
```

These two statements declare two `int` variables: `leftHandSide` and `rightHandSide`. Notice how the variables are initialized. In both cases, the `Parse` method of the `System.Int32` struct is called. (`System` is a namespace, and `Int32` is the name of the struct in this namespace. You'll learn about structs in Chapter 9, "Creating value types with enumerations and structures.") You have seen this method before; it takes a single `string` parameter and converts it to an `int` value. These two lines of code take what the user has typed into the `lhsOperand` and `rhsOperand` boxes on the form and convert it to `int` values.

4. Look at the fourth statement in the `calculateClick` method (after the `if` statement and another opening brace):

```
calculatedValue = addValues(leftHandSide, rightHandSide);
```

This statement calls the `addValues` method, passing the values of the `leftHandSide` and `rightHandSide` variables as its arguments. The value returned by the `addValues` method is stored in the `calculatedValue` variable.

5. Look at the next statement:

```
showResult(calculatedValue);
```

This statement calls the `showResult` method passing the value in the `calculatedValue` variable as its argument. The `showResult` method does not return a value.

6. In the Code and Text Editor window, find the `showResult` method you looked at earlier.

The only statement of this method is this:

```
result.Text = answer.ToString();
```

Notice that the `ToString` method call uses parentheses even though there are no arguments.



**Tip** You can call methods belonging to other objects by prefixing the method with the name of the object. In the preceding example, the expression `answer.ToString()` calls the method named `ToString` belonging to the object called `answer`.

## Returning multiple values from a method

There may be occasions when you want to return more than one value from a method. For example, in the `Methods` project, you might want to combine the effects of the `divideValues` and `remainderValues` operations into a single method that returns the results of dividing the two operands together with the remainder. You can achieve this by returning a tuple.

A tuple is simply a small collection of values (strictly speaking, a tuple contains two values, but C# tuples can comprise bigger sets than this). You indicate that a method returns a tuple by specifying a list of types as part of the method definition—one type for each value returned. The `return` statement in the method returns a list of values, as shown by the following example:

```
(int, int) returnMultipleValues(...)
{
 int val1;
 int val2;
 ... // Calculate values for val1 and val2
 return(val1, val2);
}
```

When you call the method, you provide an equivalent list of variables for holding the results:

```
int retVal1, retVal2;
(retVal1, retVal2) = returnMultipleValues(...);
```

If you're only interested in one of the return values, you can use a `discard`. (Discards were introduced in Chapter 1, "Welcome to C#.") The following code shows an example. In this case, the program is only interested in the second value in the tuple and discards the first:

```
int retVal2;
(_, retVal2) = returnMultipleValues(...);
```

The next exercise illustrates how to create and call a method that returns a tuple.

### To create and call a method that returns a tuple

1. Return to the `Methods` project and display the code for `MainPage.xaml.cs` in the Code and Text Editor window.
2. Locate the `divideValues` and `remainderValues` methods and delete them.
3. Add the following method definition in place of the methods you have just removed:

```
private (int, int) divide(int leftHandSide, int rightHandSide)
{
}
```

This method returns a tuple containing two values. These values contain the results of dividing the `leftHandSide` variable by the `rightHandSide` variable, as well as the remainder.

4. In the body of the method, add the code shown here in bold. This code performs the calculations and returns a tuple containing the results:

```
private (int, int) divide(int leftHandSide, int rightHandSide)
{
 expression.Text = $"{leftHandSide} / {rightHandSide}";
 int division = leftHandSide / rightHandSide;
 int remainder = leftHandSide % rightHandSide;
 return (division, remainder);
}
```

5. Scroll up to the `calculateClick` method and locate the following code near the end of the method:

```
else if (division.IsChecked.HasValue && division.IsChecked.Value)
{
 calculatedValue = divideValues(leftHandSide, rightHandSide);
 showResult(calculatedValue);
}
else if (remainder.IsChecked.HasValue && remainder.IsChecked.Value)
{
 calculatedValue = remainderValues(leftHandSide, rightHandSide);
 showResult(calculatedValue);
}
```

6. Delete this code. The `divideValues` and `remainderValues` methods no longer exist and have been replaced with the single `divide` method.
7. Add the following statements in place of the code you just deleted:

```
else if (division.IsChecked.HasValue && division.IsChecked.Value)
{
 int division, remainder;
 (division, remainder) = divide(leftHandSide, rightHandSide);
 result.Text = $"{division} remainder {remainder}";
}
```

This code calls the `divide` method. The values returned are displayed in the `Result` text block control.

8. In Solution Explorer, double-click the **MainPage.xaml** file to display the form in the Design View window.
9. Select the **% Remainder** option button. Then press **Delete** to remove it from the form. This option button is no longer required.
10. On the **Debug** menu, select **Start Debugging** to build and run the application.
11. In the **Left Operand** box, enter **59**; in the **Right Operand** box, enter **13**; select **Division**, and then select **Calculate**.

12. Verify that the Result box contains the message "4 remainder 7."

Methods

Left Operand

59

Right Operand

13

☐ + Addition  
☐ - Subtraction  
☐ \* Multiplication  
☒ / Division

Calculate

Expression: 59 / 13

Result: 4 remainder 7

13. Return to Visual Studio. On the **Debug** menu, select **Stop Debugging**.

## Applying scope

You create variables to hold values. You can create variables at various points in your applications. For example, the `calculateClick` method in the `Methods` project creates an `int` variable called `calculatedValue` and assigns it an initial value of 0, like this:

```
private void calculateClick(object sender, RoutedEventArgs e)
{
 int calculatedValue = 0;
 ...
}
```

This variable comes into existence at the point where it is defined, and subsequent statements in the `calculateClick` method can use it. This is an important point: A variable can be used only after it has been created. When the method has finished, this variable disappears and cannot be used elsewhere.

When a variable can be accessed at a particular location in a program, the variable is said to be in *scope* at that location. The `calculatedValue` variable has method scope; it can be accessed throughout the `calculateClick` method but not outside that method. You can also define variables with different scope—for example, you can define a variable outside a method but within a class, and this variable can be accessed by any method within that class. Such a variable is said to have *class scope*.

To put it another way, the scope of a variable is simply the region of the program in which that variable is usable. Scope applies to methods as well as variables. The scope of an identifier (of a variable or method) is linked to the location of the declaration that introduces the identifier in the program, as you will learn next.

## Defining local scope

The opening and closing braces that form the body of a method define the scope of the method. Any variables you declare inside the body of a method are scoped to that method; they disappear when the method ends and can be accessed only by code running in that method. These variables are called *local variables* because they are local to the method in which they are declared; they are not in scope in any other method.

The scope of local variables means that you cannot use them to share information between methods. Consider this example:

```
class Example
{
 void firstMethod()
 {
 int myVar;
 ...
 }

 void anotherMethod()
 {
 myVar = 42; // error - variable not in scope
 ...
 }
}
```

This code fails to compile because `anotherMethod` is trying to use the variable `myVar`, which is not in scope. The variable `myVar` is available only to statements in `firstMethod` that occur after the line of code that declares `myVar`.

## Defining class scope

The opening and closing braces that form the body of a class define the scope of that class. Any variables you declare within the body of a class (but not within a method) are scoped to that class. The proper C# term for a variable defined by a class is *field*. As mentioned, in contrast with local variables, you can use fields to share information between methods. Here's an example:

```
class Example
{
 void firstMethod()
 {
 myField = 42; // ok
 ...
 }

 void anotherMethod()
 {
 myField++; // ok
 ...
 }

 int myField = 0;
}
```



The variable `myField` is defined in the class but outside the methods `firstMethod` and `anotherMethod`. Therefore, `myField` has class scope and is available for use by all methods in that class.

There is one other point to note about this example. In a method, you must declare a variable before you can use it. Fields are a little different. A method can use a field before the statement that defines the field; the C# compiler sorts out the details for you. Notwithstanding this capability of the compiler, most programmers consider it good practice to declare fields at the start of a class, before the methods that use them; it helps to aid in understanding of the data that the methods are using.

## Overloading methods

If two identifiers have the same name and are declared in the same scope, they are said to be *overloaded*. Often an overloaded identifier is a bug that's trapped as a compile-time error. For example, if you declare two local variables with the same name in the same method, the compiler reports an error. Similarly, if you declare two fields with the same name in the same class or two identical methods in the same class, you also get a compile-time error. This fact might seem hardly worth mentioning given that the compiler catches these errors. However, there's a way that you can overload an identifier for a method that is both useful and important.

Consider the `WriteLine` method of the `Console` class. You've already used this method for writing a string to the screen. However, when you type `WriteLine` in the Code and Text Editor window when writing C# code, notice that Microsoft IntelliSense gives you 18 different options!

Each version of the `WriteLine` method takes a different set of parameters. One version takes no parameters and simply outputs a blank line. Another version takes a `bool` parameter and outputs a string representation of its value (`True` or `False`). Yet another implementation takes a `decimal` parameter and outputs it as a string, and so on. At compile time, the compiler looks at the types of the arguments you're passing in and then arranges for your application to call the version of the method that has a matching set of parameters. Here's an example:

```
static void Main()
{
 Console.WriteLine("The answer is ");
 Console.WriteLine(42);
}
```

Overloading is primarily useful when you need to perform the same operation on different data types or varying groups of information. You can overload a method when the different implementations have different sets of parameters—that is, when they have the same name but a different number of parameters or when the types of the parameters differ. When you call a method, you supply a comma-separated list of arguments, and the number and type of the arguments are used by the compiler to select one of the overloaded methods. However, keep in mind that although you can overload the parameters of a method, you can't overload the return type of a method. In other words, you can't declare two methods with the same name that differ only in their return type. (The compiler is clever, but not that clever.)

## Writing methods

In the following exercises, you'll create a method that calculates how much a consultant would charge for a given number of consultancy days at a fixed daily rate. You will start by developing the logic for the application and then use the Generate Method Stub Wizard to help you write the methods that are used by this logic. Next, you'll run these methods in a console application to get a feel for the program. Finally, you'll use the Visual Studio 2022 debugger to step into and out of the method calls as they run.

### To develop the logic for the application

1. Using Visual Studio 2022, open the **DailyRate** project, which is in the **\Microsoft Press\VC SBS\Chapter 3\DailyRate** folder in your **Documents** folder.
2. In Solution Explorer, in the **DailyRate** project, double-click the file **Program.cs** to display the code for the program in the Code and Text Editor window.

This program is simply a test harness for you to try out your code. When the application starts running, it calls the `run` method. You add the code that you want to try to the `run` method. (How the method is called requires an understanding of classes, which you look at in Chapter 7.)

3. Add the following statements shown in bold to the body of the `run` method, between the opening and closing braces:

```
void run()
{
 double dailyRate = readDouble("Enter your daily rate: ");
 int noOfDays = readInt("Enter the number of days: ");
 writeFee(calculateFee(dailyRate, noOfDays));
}
```

The block of code you have just added to the `run` method calls the `readDouble` method (which you will write shortly) to ask the user for the daily rate for the consultant. The next statement calls the `readInt` method (which you will also write) to obtain the number of days. Finally, the `writeFee` method (again, to be written by you) is called to display the results on the screen. Notice that the value passed to `writeFee` is the value returned by the `calculateFee` method, which takes the daily rate and the number of days and calculates the total fee payable.

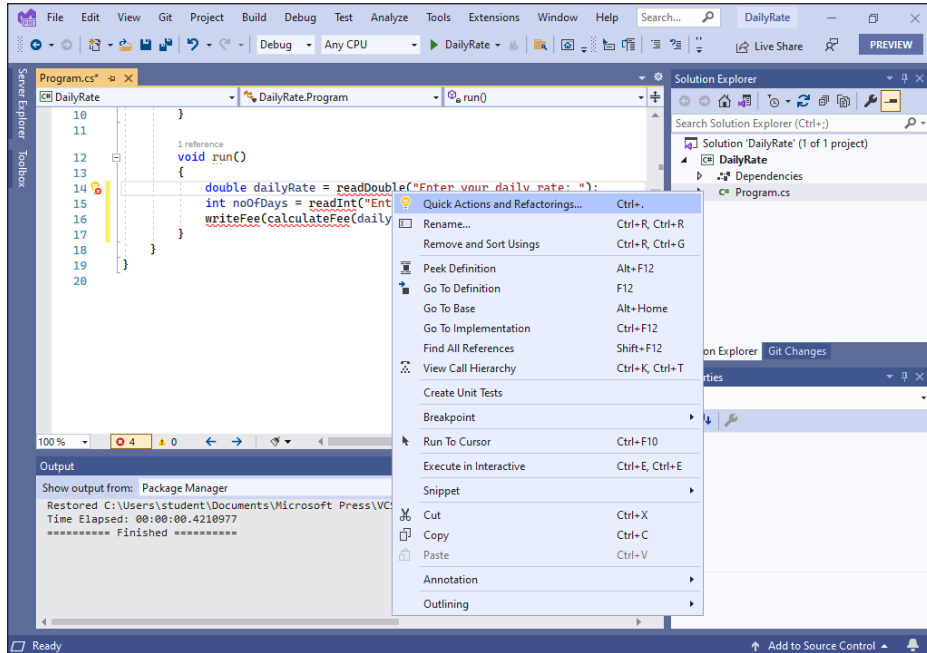


**Note** You have not yet written the `readDouble`, `readInt`, `writeFee`, and `calculateFee` methods, so IntelliSense does not display these methods when you type this code. Do not try to build the application yet, because it will fail.

## To write the methods by using the Generate Method Stub Wizard

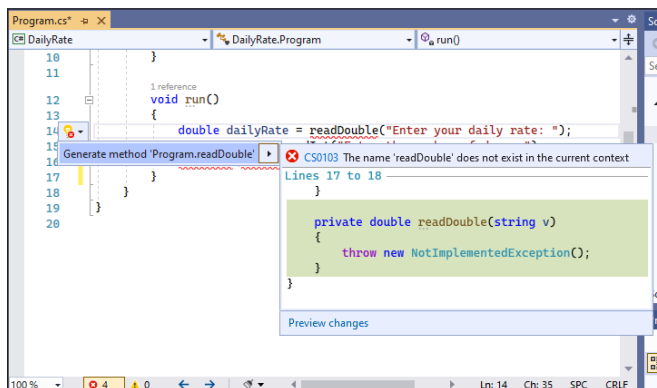
1. In the Code and Text Editor window, in the run method, right-click the `readDouble` method call.

A shortcut menu appears that contains useful commands for generating and editing code.



2. On the shortcut menu, select **Quick Actions and Refactorings**.

Visual Studio verifies that the `readDouble` method does not exist and displays a wizard that enables you to generate a stub for this method. Visual Studio examines the call to the `readDouble` method, ascertains the type of its parameters and return value, and suggests a default implementation.



3. Select **Generate Method 'Program.readDouble'**. Visual Studio adds the following method to your code:

```
private double readDouble(string v)
{
 throw new NotImplementedException();
}
```

The new method is created with the `private` qualifier, which is described in Chapter 7. The body of the method currently just throws a `NotImplementedException` exception. (Exceptions are described in Chapter 6.) You replace the body with your own code in the next step.

4. Delete the `throw new NotImplementedException();` statement from the `readDouble` method and replace it with the following lines of code shown in bold:

```
private double readDouble(string v)
{
 Console.Write(v);
 string line = Console.ReadLine();
 return double.Parse(line);
}
```

This block of code displays the string in variable `v` on the screen. This variable is the `string` parameter passed in when the method is called; it contains the message prompting the user to type in the daily rate.



**Note** The `Console.Write` method is similar to the `Console.WriteLine` statement that you used in earlier exercises, except that it does not output a newline character after the message.

The user types a value, which is read into a `string` using the `ReadLine` method and converted to a `double` using the `double.Parse` method. The result is passed back as the return value of the method call.



**Note** The `ReadLine` method is the companion method to `WriteLine`; it reads user input from the keyboard, finishing when the user presses the Enter key. The text typed by the user is passed back as the return value. The text is returned as a `string` value.

5. In the run method, right-click the call to the `readInt` method, select **Quick Actions and Refactorings**, and then select **Generate Method 'Program.readInt'**.

The `readInt` method is generated like this:

```
private int readInt(string v)
{
 throw new NotImplementedException();
}
```

6. Replace the `throw new NotImplementedException();` statement in the body of the `readInt` method with the following code shown in bold:

```
private int readInt(string v)
{
 Console.Write(v);
 string line = Console.ReadLine();
 return int.Parse(line);
}
```

This block of code is similar to the code for the `readDouble` method. The only difference is that the method returns an `int` value, so the `string` typed by the user is converted to a number using the `int.Parse` method.

7. Right-click the call to the `calculateFee` method in the run method, select **Quick Actions and Refactorings**, and then select **Generate Method 'Program.calculateFee'**.

The `calculateFee` method is generated like this:

```
private object calculateFee(double dailyRate, int noOfDays)
{
 throw new NotImplementedException();
}
```

Notice in this case that Visual Studio uses the names of the arguments passed in to generate names for the parameters. (You can, of course, change the parameter names if they're not suitable.) What's more intriguing is the type returned by the method, which is `object`. Visual Studio is unable to determine exactly which type of value should be returned by the method from the context in which it is called. The `object` type just means a "thing," and you should change it to the type you require when you add the code to the method. Chapter 7 covers the `object` type in greater detail.

8. Change the definition of the `calculateFee` method so that it returns a `double`, as shown in bold type here:

```
private double calculateFee(double dailyRate, int noOfDays)
{
 throw new NotImplementedException();
}
```

9. Replace the body of the `calculateFee` method and change it to an expression-bodied method with the following expression shown in bold; remove the curly braces and use `=>` to indicate the expression that defines the body of the method. This statement calculates the fee payable by multiplying the two parameters together:

```
private double calculateFee(double dailyRate, int noOfDays) => dailyRate * noOfDays;
```

10. Right-click the call to the `writeFee` method in the run method, select **Quick Actions and Refactorings**, and then select **Generate Method 'Program.writeFee'**.

Notice that Visual Studio uses the definition of the `writeFee` method to ascertain that its parameter should be a `double`. Also, the method call does not use a return value, so the type of the method is `void`:

```
private void writeFee(double v)
{
 ...
}
```



**Tip** If you feel sufficiently comfortable with the syntax, you can also write methods by typing them directly into the Code and Text Editor window. You do not always have to use the Generate menu option.

11. Replace the code in the body of the `writeFee` method with the following statement, which calculates the fee and adds a 10 percent commission before displaying the result. Again, notice that this is now an expression-bodied method:

```
private void writeFee(double v) => Console.WriteLine($"The consultant's fee is: {v * 1.1}");
```

12. On the **Build** menu, select **Build Solution**.

### To test the program

1. On the **Debug** menu, select **Start Without Debugging**.

Visual Studio 2022 builds the program and then runs it. A console window appears.

2. At the **Enter Your Daily Rate** prompt, type **525**. Then press **Enter**.
3. At the **Enter The Number of Days** prompt, type **17**. Then press **Enter**.

The program writes the following message to the console window:

```
The consultant's fee is: 9817.5
```

4. Press the **Enter** key to close the application and return to Visual Studio 2022.

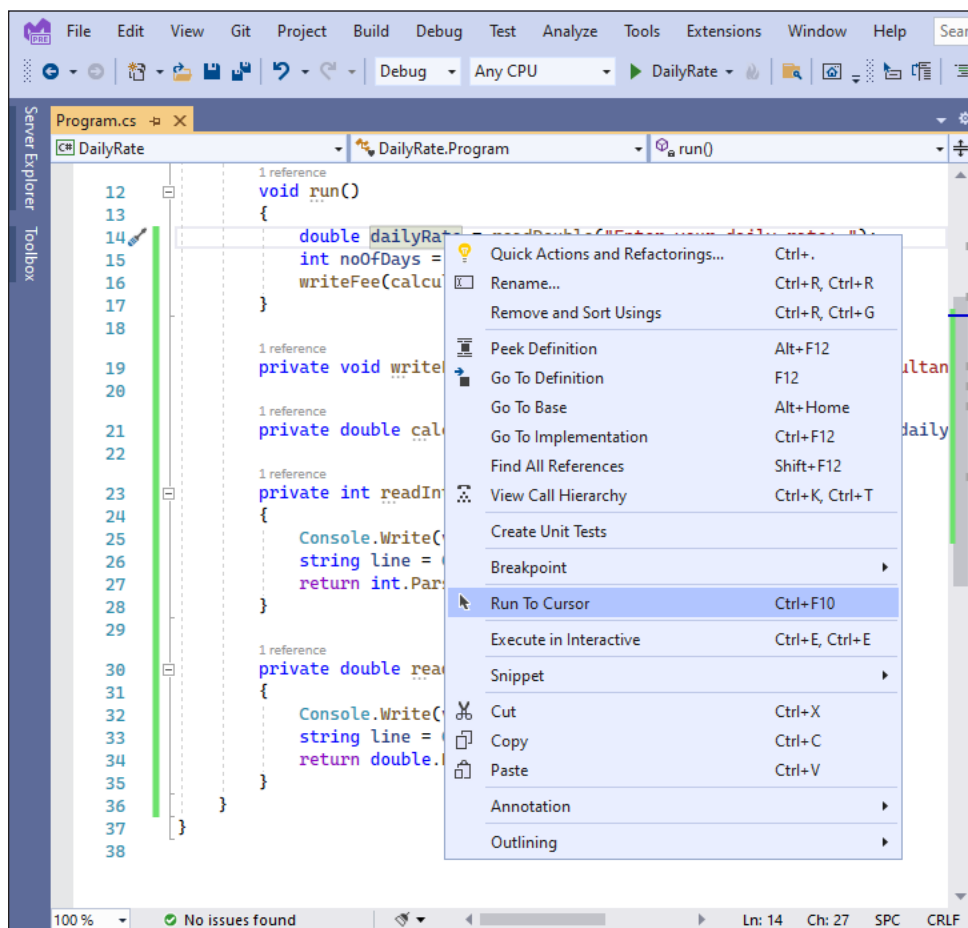
## Using the Visual Studio Debugger to step through methods

Visual Studio provides powerful debugging features that enable you to see how your application is executing on a line-by-line basis. This is useful when you need to understand why your program isn't running as expected or if you want to see how an application logic written by another developer flows through the various methods.

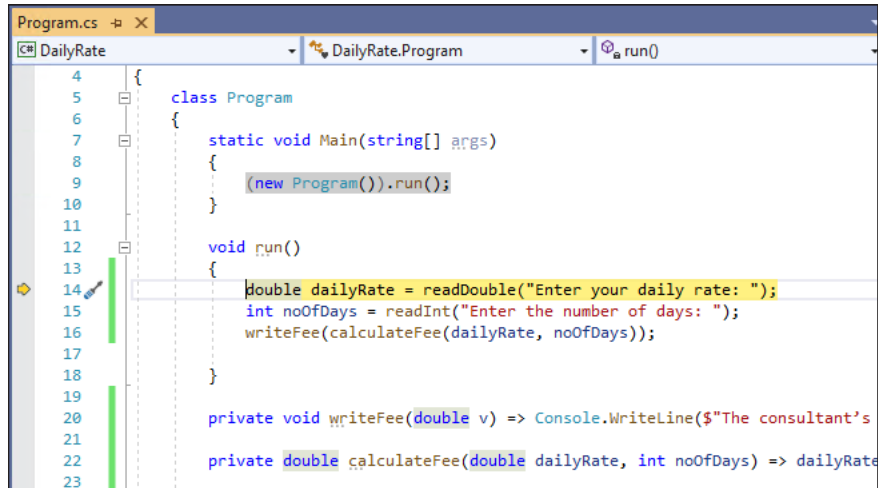
In the next exercise, you'll use the Visual Studio 2022 debugger to run your program in slow motion. You'll see when each method is called (which is referred to as *stepping into the method*) and then see how each return statement transfers control back to the caller (also known as *stepping out of the method*). While you are stepping into and out of methods, you can use the tools on the Debug toolbar. However, the same commands are also available on the Debug menu when an application is running in debug mode.

## To step through the methods by using the Visual Studio 2022 debugger

1. In the Code and Text Editor window, find the run method.
2. Move the cursor to the first statement in the run method:  
`double dailyRate = readDouble("Enter your daily rate: ");`
3. Right-click anywhere on this line, and then select **Run To Cursor**.



The program starts, runs until it reaches the first statement in the run method, and then pauses. A yellow arrow in the left margin of the Code and Text Editor window indicates the current statement, and the statement itself is highlighted with a yellow background.



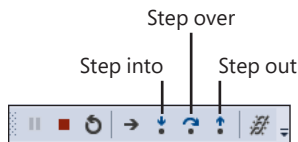
The screenshot shows a code editor window titled 'Program.cs'. The code is as follows:

```
4 {
5 class Program
6 {
7 static void Main(string[] args)
8 {
9 (new Program()).run();
10 }
11 }
12 void run()
13 {
14 double dailyRate = readDouble("Enter your daily rate: ");
15 int noOfDays = readInt("Enter the number of days: ");
16 writeFee(calculateFee(dailyRate, noOfDays));
17 }
18
19 private void writeFee(double v) => Console.WriteLine($"The consultant's
20
21 private double calculateFee(double dailyRate, int noOfDays) => dailyRate
22
23
```

The 'run()' method and its first statement are highlighted with a yellow background. A yellow arrow in the left margin points to the first statement of the 'run()' method.

4. On the **View** menu, point to **Toolbars**, and then ensure that the **Debug** toolbar is selected.

If it wasn't already visible, the Debug toolbar opens. It might appear docked with the other toolbars. If you can't see the toolbar, try using the Toolbars command on the View menu to hide it and look to see which buttons disappear. Then, display the toolbar again. The Debug toolbar looks like this:



5. On the Debug toolbar, select the **Step Into** button. (This is the fifth button from the left on the Debug toolbar.)

The debugger steps into the method being called. The yellow cursor jumps to the opening brace at the start of the `readDouble` method.

6. Select **Step Into** again to advance the cursor to the first statement:

```
Console.Write(v);
```



**Tip** You can press F11 instead of selecting Step Into on the Debug toolbar.

7. On the Debug toolbar, select **Step Over**. (This is the sixth button from the left.)



The method executes the next statement without debugging it (stepping into it). This action is useful primarily if the statement calls a method, but you don't want to step through every statement in that method. The yellow cursor moves to the second statement of the method, and the program displays the Enter Your Daily Rate prompt in a console window before returning to Visual Studio 2022. (The console window might be hidden behind Visual Studio.)



**Tip** You can press F10 instead of selecting Step Over on the Debug toolbar.

8. On the Debug toolbar, select **Step Over** again.

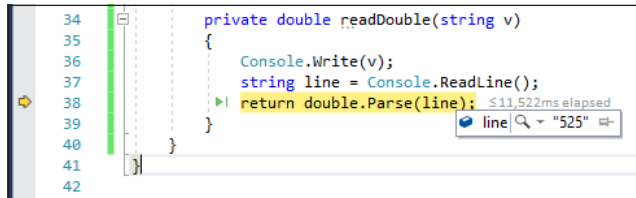
This time, the yellow cursor disappears, and the console window gets the focus because the program is executing the `Console.ReadLine` method and is waiting for you to type something.

9. Type **525** in the console window. Then press **Enter**.

Control returns to Visual Studio 2022. The yellow cursor appears on the third line of the method.

10. Hover the cursor over the reference to the `line` variable on either the second or third line of the method. (It doesn't matter which.)

A ScreenTip appears, displaying the current value of the `line` variable ("525"). You can use this feature to ensure that a variable has been set to an expected value while you step through methods.



11. On the Debug toolbar, select **Step Out**. (This is the seventh button from the left.)

The current method continues to run uninterrupted to its end. The `readDouble` method finishes, and the yellow cursor is placed back at the first statement of the run method. This statement has now finished running.



**Tip** You can press Shift+F11 instead of selecting Step Out on the Debug toolbar.

12. On the Debug toolbar, select **Step Into**.

The yellow arrow (and highlighting) moves to the second statement in the run method:

```
int noOfDays = readInt("Enter the number of days: ");
```

13. On the Debug toolbar, select **Step Over**.

This time, you have chosen to run the method without stepping through it. The console window appears again, prompting you for the number of days.

14. In the console window, type **17**. Then press **Enter**.

Control returns to Visual Studio 2022. (You might need to bring Visual Studio to the foreground.) The yellow cursor moves to the third statement of the run method:

```
wri teFee(calcu lateFee(dailyRate, noOfDays));
```

15. On the Debug toolbar, select **Step Into**.

The yellow cursor jumps to the expression that defines the body of the `calcu lateFee` method. This method is called first, before `wri teFee`, because the value returned by this method is used as the parameter to `wri teFee`.

16. On the Debug toolbar, select **Step Out**.

The `calcu lateFee` method call completes, and the yellow arrow jumps back to the third statement of the run method.

17. On the Debug toolbar, select **Step Into** again.

This time, the yellow arrow jumps to the statement that defines the body of the `wri teFee` method.

18. Place the mouse over the `v` parameter in the method definition.

The value of `v`, 8925, is displayed in a ScreenTip.

19. On the Debug toolbar, select **Step Out**.

The message “The consultant’s fee is: 9817.5” is displayed in the console window. (You might need to bring the console window to the foreground to display it if it’s hidden behind Visual Studio 2022.) The yellow arrow returns to the third statement in the run method.

20. On the toolbar, select **Continue** to continue running the program without stopping at each subsequent statement.



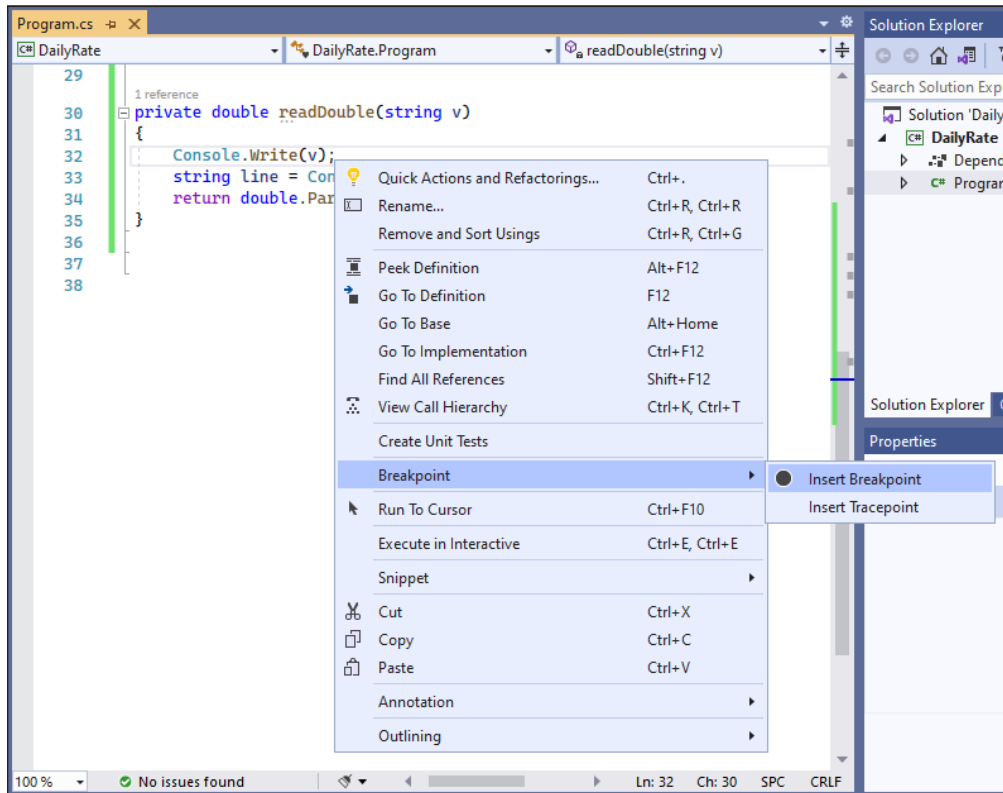
**Tip** If the Continue button is not visible, select the Add Or Remove Buttons drop-down menu that appears at the end of the Debug toolbar and select Continue. The Continue button should now appear. Alternatively, you can press F5 to continue running the application without debugging.

The application completes and finishes running. Notice that the Debug toolbar disappears when the application finishes; by default, the Debug toolbar is displayed only when you’re running an application in debug mode.

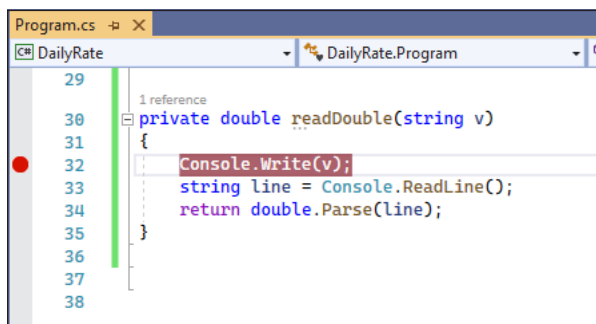
Another powerful feature of the debugger is the ability to modify the code being run at debug time; that is, you don’t need to stop the application and restart it. This approach can substantially reduce the time required to debug applications. In the next exercise, you’ll see a simple example of this technique in action.

## To amend code during a debug session

1. In the Code and Text Editor window, find the `readDouble` method.
2. Right-click the `Console.WriteLine(v);` statement, point to **Breakpoint**, and then select **Insert Breakpoint**. When you run the application in debug mode, the application will stop when it reaches this statement and drop into the debugger.



After adding the breakpoint, the statement appears highlighted in the Code and Text Editor window, together with a large red dot in the margin.



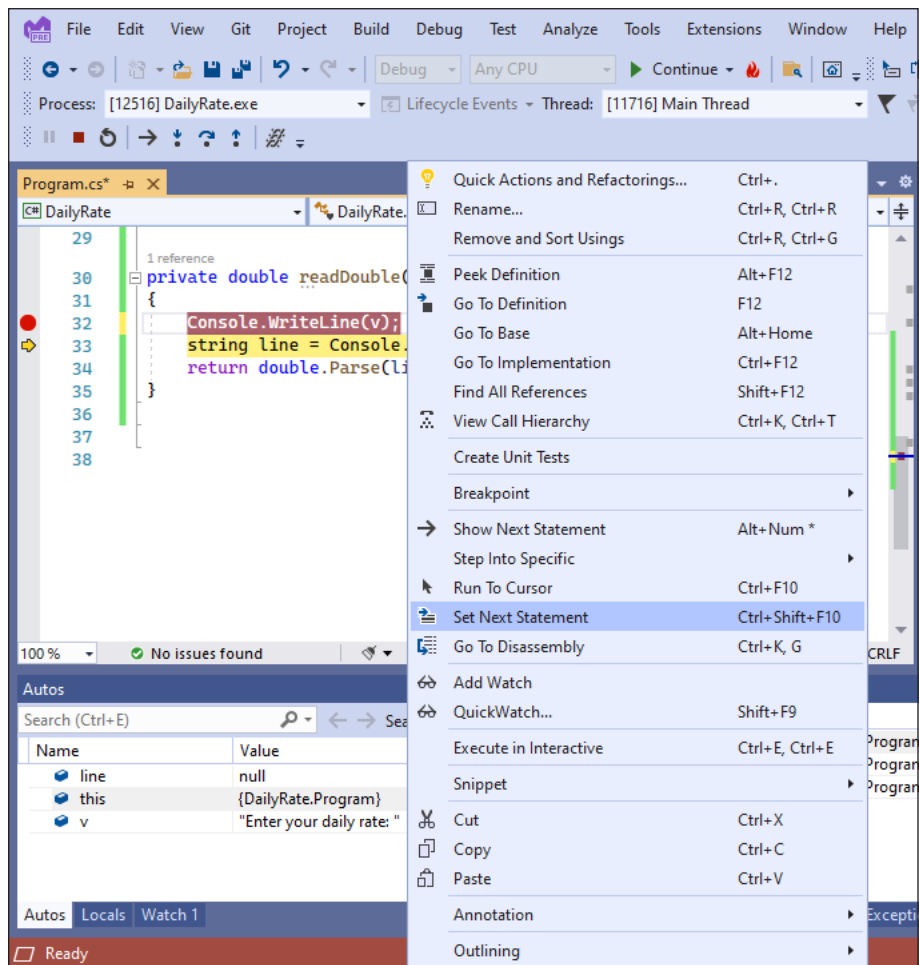
3. In the Visual Studio toolbar, in the **Debug** menu, select **Start Debugging**.

The application starts, but halts when it reaches the statement marked as the breakpoint.

4. In the Debug toolbar, select **Step Over** to allow this statement to execute.
5. Switch to the console window for the application. It should display the message “Enter your daily rate:” and the user is prompted to enter a value on the same line.

You decide that you want to change the way this message is displayed. You want to add a new-line character after the message.

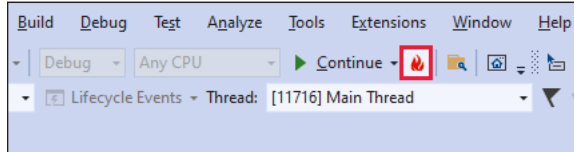
6. Switch back to Visual Studio currently running the application in debug mode.
7. Change the statement `Console.Write(v);` to `Console.WriteLine(v);` without stopping the application.
8. Right-click the amended statement, and then select **Set Next Statement**.



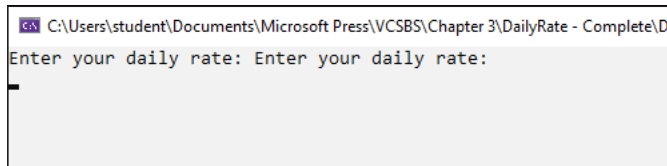
This action causes the debugger to return to that statement, so when you select Step Over, this statement will be repeated.



**Note** If you are building a UWP app (as opposed to a Console app), you can also modify the XAML layout for the user interface during a debug session. To do this, make the necessary changes to the XAML description while the app is suspended at a breakpoint. Then, in the Visual Studio toolbar, select the Apply Code Changes button. This button has a flame icon to indicate it causes a hot reload of the UI.



9. On the Debug toolbar, select **Step Over**. The amended statement should now run.
10. Switch back to the console window for the application. It will still display the original message, followed by "Enter your daily rate:" again. However, the cursor waiting for the user input is now on the next line down because the `Console.WriteLine` statement has added a newline character.



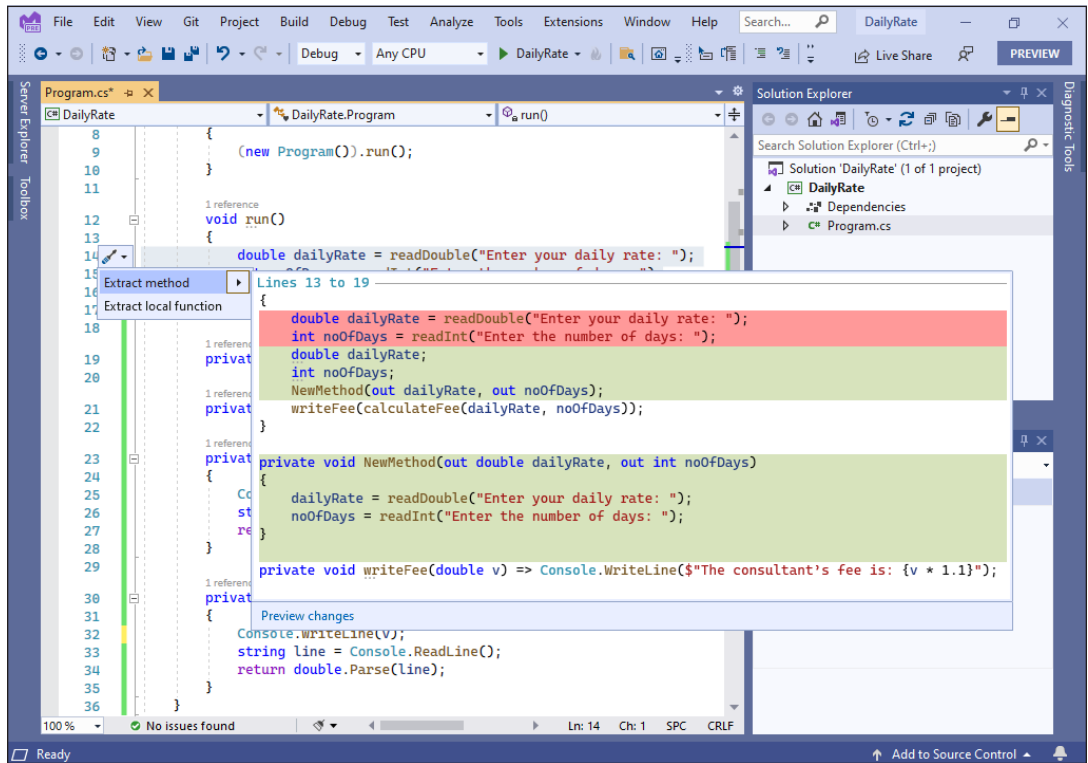
11. Return to Visual Studio and select **Continue** in the toolbar.
12. Enter the same values that you did previously and allow the application to run to completion. The result should be the same as before.
13. On the **Debug** menu, select **Delete All Breakpoints** to remove the breakpoint from the `readDouble` method.
14. In the Visual Studio message box that appears, select **Yes** to confirm that you want to remove breakpoints.

## Refactoring code

A very useful feature of Visual Studio 2022 is the ability to refactor code. This can make your code more readable and maintainable by breaking it out into smaller, well-defined pieces.

Occasionally, you will find yourself writing the same (or similar) code in more than one place in an application. When this occurs, you can highlight and right-click the block of code you just typed, select **Quick Actions and Refactoring**, and choose **Extract Method**. The selected code is moved to a new method named `NewMethod`.

The Extract Method Wizard can determine whether the method should take any parameters and return a value. After the method has been generated, you should change its name (by overtyping) to something meaningful and also change the statement that has been generated to call this method with the new name.



## Nesting methods

In addition to refactoring, sometimes you want to break down a large method into smaller pieces. You can implement each piece as a helper method in its own right. This helps you test methods that perform complex processes and verify that each part of the large method functions as expected before bringing them together. It can also aid readability and make a large method easier to maintain.



**Note** The terms *large method* and *helper method* are not official vocabulary in C#. I've used them in this discussion to distinguish between a method that's broken down into smaller pieces (the large method) and the methods that implement these smaller pieces (the helper methods).

By default, methods (large and helper) are accessible across the class in which they are defined and can be invoked from any other methods in that class. In the case of helper methods that are used only by one large method, it can make sense to keep these methods local to the large method that runs them. This approach can ensure that a helper method designed to operate in a given context is

not used accidentally by another method for which it was not designed. This is also good practice for implementing encapsulation; the inner workings of a large method, including the helper methods that it invokes, can be kept separate from other methods. This practice reduces any dependencies between large methods; you can safely change the implementation of a large method and the helper methods that it invokes without accidentally affecting other elements of your application.

You can create helper methods by nesting them inside the large method that uses them, as shown in the next exercise. This exercise calculates factorials. You can use factorials to determine how many ways you can arrange a given number of items. The factorial of a positive integer,  $n$ , is defined in a recursive manner as  $n * \text{factorial}(n - 1)$ , where the factorial of 1 is 1. For example, the factorial of 3 is  $3 * \text{factorial}(2)$ , which is in turn  $2 * \text{factorial}(1)$ , which is 1. This calculation evaluates as  $3 * 2 * 1$ , or 6. If you have three items in a set, you can arrange them in six different ways. Similarly, if you have four items, you can arrange them in 24 different ways ( $4 * \text{factorial}(3)$ ), and you can arrange five items in 120 different ways ( $5 * \text{factorial}(4)$ ).

### To calculate factorials

1. Start **Visual Studio 2022** if it's not already running.
2. Open the **Factorial** solution, which is in the `\Microsoft Press\VCSBS\Chapter 3\Factorial` folder in your **Documents** folder.
3. In Solution Explorer, in the Factorials project, double-click the file **Program.cs** to display the code for the program in the Code and Text Editor window.
4. Add the following statements shown in bold to the body of the run method, between the opening and closing braces:

```
void run()
{
 Console.Write("Please enter a positive integer: ");
 string inputValue = Console.ReadLine();
 long factorialValue = CalculateFactorial(inputValue);
 Console.WriteLine($"Factorial({inputValue}) is {factorialValue}");
}
```

This code prompts the user to enter a numeric value and then calls the `CalculateFactorial` function (which you will write next) with this value before displaying the result.

5. Add a new method named `CalculateFactorial` below the run method. This method should take a string parameter named `input` and return a long integer value, as follows:

```
long CalculateFactorial(string input)
{
}
```

6. In the `CalculateFactorial` method, after the initial opening brace, add the statement shown here in bold:

```
long CalculateFactorial(string input)
{
 int inputValue = int.Parse(input);
}
```

7. This statement converts the `string` value passed in as the parameter to an integer. (The code does not currently check to make sure that the user has entered a valid integer; you will see how to do this in Chapter 6.)
8. Add a nested method named `factorial` to the `CalculateFactorial` function. The `factorial` method should take an `int` value and return a `long`. You'll use this method to calculate the factorial of the input parameter:

```
long CalculateFactorial(string input)
{
 int inputValue = int.Parse(input);
 long factorial (int dataValue)
 {
 }
```

9. In the body of the `factorial` method, add the statements shown here in bold. This code calculates the factorial of the input parameter using the recursive algorithm described earlier:

```
long CalculateFactorial(string input)
{
 int inputValue = int.Parse(input);
 long factorial (int dataValue)
 {
 if (dataValue == 1)
 {
 return 1;
 }
 else
 {
 return dataValue * factorial(dataValue - 1);
 }
 }
}
```

10. In the `CalculateFactorial` method, call the `factorial` method using the integer value provided as input and return the result:

```
long CalculateFactorial(string input)
{
 int inputValue = int.Parse(input);
 long factorial (int dataValue)
 {
 if (dataValue == 1)
 {
 return 1;
 }
 else
 {
 return dataValue * factorial(dataValue - 1);
 }
 }

 long factorialValue = factorial(inputValue);
 return factorialValue;
}
```



11. On the Debug menu, select **Start Without Debugging**.

Visual Studio 2022 builds the program and then runs it. A console window appears.

12. At the **Please Enter a Positive Integer** prompt, type **4**, and then press **Enter**.

The program writes the following message to the console window:

```
Factorial(4) is 24
```

13. Press the **Enter** key to close the application and return to Visual Studio 2022.

14. Run the application and enter the value **5** when prompted. This time, the application should display the following message:

```
Factorial(5) is 120
```

15. Experiment with other values. Note that if you enter an input value that's too large (try 60, for example), the result will exceed that of the range that can be stored in a long integer, and you will get an incorrect result—most likely a negative number generated as a result of numeric overflow. You'll learn more about how to handle this eventuality by using checked exceptions in Chapter 6.

## Using optional parameters and named arguments

---

By defining overloaded methods, you can implement different versions of a method that take different parameters. When you build an application that uses overloaded methods, the compiler determines which specific instances of each method it should use to satisfy each method call. This is a common feature of many object-oriented languages, not just C#.

However, developers can use other languages and technologies to build Windows applications and components that do not follow these rules. A key feature of C# and other languages designed for .NET is the ability to interoperate with applications and components written with other technologies. One of the principal technologies that underpins many Windows applications and services running outside the .NET environment is the Component Object Model (COM). In fact, the common language runtime (CLR) used by .NET is also heavily dependent on COM, as is the Windows Runtime of Windows. COM does not support overloaded methods; instead, it uses methods that can take optional parameters. To make it easier to incorporate COM libraries and components into a C# solution, C# supports optional parameters.

Optional parameters are also useful in other situations. They provide a compact and simple solution when it is not possible to use overloading because the types of the parameters do not vary sufficiently to enable the compiler to distinguish between implementations. For example, consider the following method:

```
public void DoWorkWithData(int intData, float floatData, int moreIntData) {
 ...
}
```

The `DoWorkWithData` method takes three parameters: two `ints` and a `float`. Now suppose you want to provide an implementation of `DoWorkWithData` that takes only two parameters: `intData` and `floatData`. You can overload the method like this:

```
public void DoWorkWithData(int intData, float floatData)
{
 ...
}
```

If you write a statement that calls the `DoWorkWithData` method, you can provide either two or three parameters of the appropriate types, and the compiler uses the type information to determine which overload to call:

```
int arg1 = 99;
float arg2 = 100.0F;
int arg3 = 101;
DoWorkWithData(arg1, arg2, arg3); // Call overload with three parameters
DoWorkWithData(arg1, arg2); // Call overload with two parameters
```

However, suppose that you want to implement two additional versions of `DoWorkWithData` that take only the first parameter and the third parameter. You might be tempted to try this:

```
public void DoWorkWithData(int intData)
{
 ...
}
public void DoWorkWithData(int moreIntData)
{
 ...
}
```

The issue here is that these two overloads appear identical to the compiler. Your code will fail to compile and will instead generate the error “Type ‘*typename*’ already defines a member called ‘`DoWorkWithData`’ with the same parameter types.” To understand why this is so, think what would happen if this code were legal. Consider the following statements:

```
int arg1 = 99;
int arg3 = 101;
DoWorkWithData(arg1);
DoWorkWithData(arg3);
```

Which overload or overloads would the calls to `DoWorkWithData` invoke? Using optional parameters and named arguments can help to solve this problem.

## Defining optional parameters

You specify that a parameter is optional when you define a method by providing a default value for the parameter. You indicate a default value by using the assignment operator. In the `optMethod` method shown next, the first parameter is mandatory because it does not specify a default value, but the second and third parameters are optional:

```
void optMethod(int first, double second = 0.0, string third = "Hello") {
 ...
}
```

You must specify all mandatory parameters before any optional parameters.

You can call a method that takes optional parameters in the same way that you call any other method: You specify the method name and provide any necessary arguments. The difference with methods that take optional parameters is that you can omit the corresponding arguments and the method will use the default value when it runs. In the example that follows, the first call to the `optMethod` method provides values for all three parameters. The second call specifies only two arguments, and these values are applied to the first and second parameters. The third parameter receives the default value of "Hello" when the method runs.

```
optMethod(99, 123.45, "World"); // Arguments provided for all three parameters
optMethod(100, 54.321); // Arguments provided for first two parameters only
```

## Passing named arguments

By default, C# uses the position of each argument in a method call to determine which parameter the argument applies to. Hence, the second example of the `OptMethod` method shown in the previous section passes the two arguments to the first and second parameters in the method because this is the order in which they occur in the method declaration.

With C#, you can also specify parameters by name. This feature lets you pass the arguments in a different sequence. To pass an argument as a named parameter, you specify the name of the parameter, followed by a colon and the value to use. The following examples perform the same function as those shown in the previous section, except that the parameters are specified by name:

```
optMethod(first : 99, second : 123.45, third : "World");
optMethod(first : 100, second : 54.321);
```

Named arguments give you the ability to pass arguments in any order. You can rewrite the code that calls the `optMethod` method as shown here:

```
optMethod(third : "World", second : 123.45, first : 99); optMethod(second : 54.321, first :
100);
optMethod(third : "World", second : 123.45, first : 99);
optMethod(second : 54.321, first : 100);
```

This feature also makes it possible for you to omit arguments. For example, you can call the `optMethod` method and specify values for the first and third parameters only and use the default value for the second parameter, like this:

```
optMethod(first : 99, third : "World");
```

Additionally, you can mix positional and named arguments. However, if you use this technique, you must specify all the positional arguments before the first named argument.

```
optMethod(99, third : "World"); // First argument is positional
```

## Resolving ambiguities with optional parameters and named arguments

Using optional parameters and named arguments can result in some possible ambiguities in your code. You must understand how the compiler resolves these ambiguities; otherwise, you might find your applications behaving in unexpected ways.

Suppose you define the `optMethod` method as an overloaded method, as shown in the following example:

```
void optMethod(int first, double second = 0.0, string third = "Hello")
{
 ...
}

void optMethod(int first, double second = 1.0, string third = "Goodbye", int fourth = 100)
{
 ...
}
```

This is perfectly legal C# code that follows the rules for overloaded methods. The compiler can distinguish between the methods because they have different parameter lists. However, as demonstrated in the following example, a problem can arise if you attempt to call the `optMethod` method and omit some of the arguments corresponding to one or more of the optional parameters:

```
optMethod(1, 2.5, "World");
```

Again, this is perfectly legal code, but which version of the `optMethod` method does it run? The answer is the version that most closely matches the method call, so the code invokes the method that takes three parameters and not the version that takes four. That makes good sense, so consider this one:

```
optMethod(1, fourth : 101);
```

In this code, the call to `optMethod` omits arguments for the `second` and `third` parameters, but it specifies the `fourth` parameter by name. Only one version of `optMethod` matches this call, so this is not a problem. This next example will get you thinking, though:

```
optMethod(1, 2.5);
```

This time, neither version of the `optMethod` method exactly matches the list of arguments provided. Both versions of the `optMethod` method have optional parameters for the `second`, `third`, and `fourth` arguments. So, does this statement call the version of `optMethod` that takes three parameters and use the default value for the `third` parameter, or does it call the version of `optMethod` that takes four parameters and use the default value for the `third` and `fourth` parameters? The answer is that it does neither. This is an unresolvable ambiguity, and the compiler does not let you compile the application.

The same situation arises with the same result if you try to call the `optMethod` method, as shown in any of the following statements:

```
optMethod(1, third : "World");
optMethod(1);
optMethod(second : 2.5, first : 1);
```

In the final exercise in this chapter, you will revisit the `DailyRate` project and practice implementing methods that take optional parameters and calling them by using named arguments. You will also test common examples of how the C# compiler resolves method calls that involve optional parameters and named arguments.

### To define and call a method that takes optional parameters

1. Using Visual Studio 2022, open the `DailyRate` solution, which is in the `\Microsoft Press\VC SBS\Chapter 3\DailyRate Using Optional Parameters` folder in your **Documents** folder.
2. In Solution Explorer, in the `DailyRate` project, double-click the file **Program.cs** to display the code for the program in the Code and Text Editor window.

This version of the application is empty apart from the `Main` method and the skeleton version of the `run` method.

3. In the `Program` class, add the `calculateFee` method after the `run` method. This is the same version of the method that you implemented in the previous set of exercises, except it takes two optional parameters with default values. The method also prints a message indicating the version of the `calculateFee` method that was called. (You will add overloaded implementations of this method in the following steps.)

```
private double calculateFee(double dailyRate = 500.0, int noOfDays = 1)
{
 Console.WriteLine("calculateFee using two optional parameters");
 return dailyRate * noOfDays;
}
```

4. Add another implementation of the `calculateFee` method to the `Program` class, as shown in the following code. This version takes one optional parameter, `dailyRate`, of type `double`. The body of the method calculates and returns the fee for a single day only.

```
private double calculateFee(double dailyRate = 500.0)
{
 Console.WriteLine("calculateFee using one optional parameter");

 int defaultNoOfDays = 1;
 return dailyRate * defaultNoOfDays;
}
```

5. Add a third implementation of the `calculateFee` method to the `Program` class. This version takes no parameters and uses hard-coded values for the daily rate and number of days.

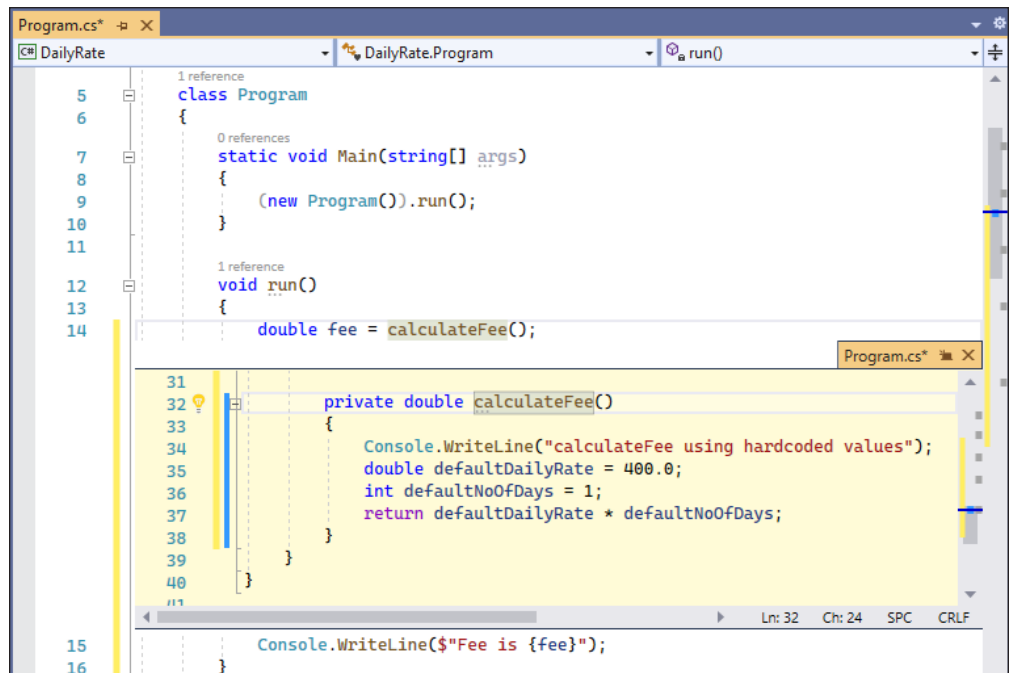
```
private double calculateFee()
{
 Console.WriteLine("calculateFee using hardcoded values");
 double defaultDailyRate = 400.0;
 int defaultNoOfDays = 1;
 return defaultDailyRate * defaultNoOfDays;
}
```

6. At the beginning of the `run` method, add the following statements in bold that call `calculateFee` and display the results:

```
public void run()
{
 double fee = calculateFee();
 Console.WriteLine($"Fee is {fee}");
}
```



**Tip** You can quickly view the definition of a method from the statement that invokes it. To do so, right-click the method call and then select **Peek Definition**. The following image shows the Peek Definition window for the `calculateFee` method. This feature is extremely useful if your code is split across multiple files, or even if it is in the same file, but the file is very long.



7. On the **Debug** menu, select **Start Without Debugging** to build and run the program.

The program runs in a console window and displays the following messages:

```
calculateFee using hardcoded values
Fee is 400
```

The run method called the version of `calculateFee` that takes no parameters rather than either of the implementations that take optional parameters because that version most closely matches the method call.

Press any key to close the console window and return to Visual Studio.

8. In the run method, modify the statement that calls `calculateFee` to match the code shown in bold here:

```
public void run()
{
 double fee = calculateFee(650.0);
 Console.WriteLine($"Fee is {fee}");
}
```

9. On the **Debug** menu, select **Start Without Debugging** to build and run the program.

The program displays the following messages:

```
calculateFee using one optional parameter
Fee is 650
```

This time, the run method called the version of `calculateFee` that takes one optional parameter. As before, this is the version that most closely matches the method call.

10. Press any key to close the console window and return to Visual Studio.
11. In the run method, modify the statement that calls `calculateFee` again, as in the following bold code:

```
public void run()
{
 double fee = calculateFee(500.0, 3);
 Console.WriteLine($"Fee is {fee}");
}
```

12. On the **Debug** menu, select **Start Without Debugging** to build and run the program.

The program displays the following messages:

```
calculateFee using two optional parameters
Fee is 1500
```

As you might expect from the previous two cases, the run method called the version of `calculateFee` that takes two optional parameters.

13. Press any key to close the console window and return to Visual Studio.

- 14.** In the run method, modify the statement that calls `calculateFee` and specify the `dailyRate` parameter by name:

```
public void run()
{
 double fee = calculateFee(dailyRate : 375.0);
 Console.WriteLine($"Fee is {fee}");
}
```

- 15.** On the **Debug** menu, select **Start Without Debugging** to build and run the program.

The program displays the following messages:

```
calculateFee using one optional parameter
Fee is 375
```

As earlier, the run method calls the version of `calculateFee` that takes one optional parameter. Changing the code to use a named argument does not change how the compiler resolves the method call in this example.

- 16.** Press any key to close the console window and return to Visual Studio.
- 17.** In the run method, modify the statement that calls `calculateFee` and specify the `noOfDays` parameter by name.

```
public void run()
{
 double fee = calculateFee(noOfDays : 4);
 Console.WriteLine($"Fee is {fee}");
}
```

- 18.** On the **Debug** menu, select **Start Without Debugging** to build and run the program.

The program displays the following messages:

```
calculateFee using two optional parameters
Fee is 2000
```

This time, the run method called the version of `calculateFee` that takes two optional parameters. The method call has omitted the first parameter (`dailyRate`) and specified the second parameter by name. The version of the `calculateFee` method that takes two optional parameters is the only one that matches the call.

- 19.** Press any key to close the console window and return to Visual Studio.
- 20.** Modify the implementation of the `calculateFee` method that takes two optional parameters. Change the name of the first parameter to `theDailyRate` and update the return statement to match that shown in bold in the following code:

```
private double calculateFee(double theDailyRate = 500.0, int noOfDays = 1)
{
 Console.WriteLine("calculateFee using two optional parameters");
 return theDailyRate * noOfDays;
}
```



21. In the `run` method, modify the statement that calls `calculateFee` and specify the `theDailyRate` parameter by name.

```
public void run()
{
 double fee = calculateFee(theDailyRate : 375.0);
 Console.WriteLine("Fee is {fee}");
}
```

22. On the **Debug** menu, select **Start Without Debugging** to build and run the program.

The program displays the following messages:

```
calculateFee using two optional parameters
Fee is 375
```

The last time you specified the fee but not the daily rate (step 14), the `run` method called the version of `calculateFee` that takes one optional parameter. This time, the `run` method called the version of `calculateFee` that takes two optional parameters. In this case, using a named argument has changed how the compiler resolves the method call. If you specify a named argument, the compiler compares the argument name to the names of the parameters specified in the method declarations and selects the method that has a parameter with a matching name. If you had specified the argument as `aDailyRate: 375.0` in the call to the `calculateFee` method, the program would have failed to compile because no version of the method has a parameter that matches this name.

23. Press any key to close the console window and return to Visual Studio.

## Summary

---

In this chapter, you learned how to define methods to implement a named block of code. You saw how to pass parameters into methods and how to return data from methods. You also saw how to call a method, pass arguments, and obtain a return value. You learned how to define overloaded methods with different parameter lists, and you saw how the scope of a variable determines where it can be accessed. Then, you used the Visual Studio 2022 debugger to step through code as it runs. Finally, you learned how to write methods that take optional parameters and how to call methods by using named parameters.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 4, “Using decision statements.”
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

## Quick reference

To	Do this
Declare a method	Write the method within a class. Specify the method name, parameter list, and return type, followed by the body of the method between braces. For example: <code>int addValues(int leftHandSide, int rightHandSide) { ... }</code>
Return a value from within a method	Write a return statement within the method. For example: <code>return leftHandSide + rightHandSide;</code>
Return multiple values from within a method	Write a return statement that returns a tuple. For example: <code>return (division, remainder);</code>
Return from a method before the end of the method	Write a return statement within the method. For example: <code>return;</code>
Define an expression-bodied method	Use the <code>=&gt;</code> sequence followed by the expression that defines the body of the method and a closing semicolon. For example: <code>double calculateFee(double dailyRate, int noOfDays) =&gt; dailyRate * noOfDays;</code>
Call a method	Write the name of the method followed by any arguments between parentheses. For example: <code>addValues(39, 3);</code>
Call a method that returns a tuple	Invoke the method as shown previously, but assign the result to a set of variables enclosed in parentheses. There should be one variable for each value of the tuple being returned. For example: <code>(division, remainder) = divide(leftHandSide, rightHandSide);</code>
Use the Generate Method Stub Wizard	Right-click a call to the method, and then select Generate Method Stub.
Create a nested method	Define the method within the body of another method. For example: <pre>long CalculateFactorial(string input) {     ...     long factorial (int dataValue)     {         if (dataValue == 1)         {             return 1;         }         else         {             return dataValue * factorial(dataValue - 1);         }     }     ... }</pre>
Display the Debug toolbar	On the View menu, point to Toolbars, and then select Debug.
Step into a method	On the Debug toolbar, select Step Into. or On the Debug menu, select Step Into.
Step out of a method	On the Debug toolbar, select Step Out. or On the Debug menu, select Step Out.
Specify an optional parameter to a method	Provide a default value for the parameter in the method declaration. For example: <code>void optMethod(int first, double second = 0.0, string third = "Hello") { ... }</code>
Pass a method argument as a named parameter	Specify the name of the parameter in the method call. For example: <code>optMethod(first : 100, third : "World");</code>



# Using decision statements

**After completing this chapter, you will be able to:**

- Declare Boolean variables.
- Use Boolean operators to create expressions whose outcome is either true or false.
- Write `if` statements to make decisions based on the result of a Boolean expression.
- Write `switch` statements to make more complex decisions.

Chapter 3, “Writing methods and applying scope,” showed how to group related statements into methods. It also demonstrated how to use parameters to pass information to a method and how to use `return` statements to pass information out of a method.

Dividing a program into a set of discrete methods, each designed to perform a specific task or calculation, is a necessary design strategy. Many programs need to solve large and complex problems. Breaking up a program into methods helps you understand these problems and focus on how to solve them, one piece at a time.

The methods in Chapter 3 are very straightforward, with each statement executing sequentially after the previous statement completes. However, to solve many real-world problems, you also must be able to write code that selectively performs different actions and that takes different paths through a method depending on the circumstances. In this chapter, you’ll learn how to accomplish this task.

## Declaring Boolean variables

---

In the world of C# programming (unlike in the real world), everything is black or white, right or wrong, true or false. For example, if you create an integer variable called `x`, assign the value 99 to it, and then ask whether `x` contains the value 99, the answer is definitely true. If you ask if `x` is less than 10, the answer is definitely false. These are examples of Boolean expressions. A Boolean expression always evaluates to true or false.



**Note** The answers to these questions are not necessarily definitive for all other programming languages. An unassigned variable has an undefined value, and you cannot, for example, say that it is definitely less than 10. Issues such as this one are a common source of errors in C and C++ programs. The Microsoft Visual C# compiler solves this problem by ensuring that you always assign a value to a variable before examining it. If you try to examine the contents of an unassigned variable, your program will not compile.

Visual C# provides a data type called `bool`. A `bool` variable can hold one of two values: `true` or `false`. For example, the following three statements declare a `bool` variable called `areYouReady`, assign `true` to that variable, and then write its value to the console:

```
bool areYouReady;
areYouReady = true;
Console.WriteLine(areYouReady); // writes True to the console
```

## Using Boolean operators

A Boolean operator performs a calculation whose result is either `true` or `false`. C# has several very useful Boolean operators, the simplest of which is the NOT operator, represented by the exclamation point (`!`). The `!` operator negates a Boolean value, yielding the opposite of that value. In the preceding example, if the value of the variable `areYouReady` is `true`, the value of the expression `!areYouReady` is `false`.

## Understanding equality and relational operators

Two Boolean operators that you'll use frequently are equality (`==`) and inequality (`!=`). These are binary operators with which you can determine whether one value is the same as another value of the same type, yielding a Boolean result. The following table summarizes how these operators work, using an `int` variable called `age` as an example:

Operator	Meaning	Example	Outcome if age is 42
<code>==</code>	Equal to	<code>age == 100</code>	false
<code>!=</code>	Not equal to	<code>age != 0</code>	true

Don't confuse the equality operator `==` with the assignment operator `=`. The expression `x==y` compares `x` with `y` and has the value `true` if the values are the same. The expression `x=y` assigns the value of `y` to `x` and returns the value of `y` as its result.

Closely related to `==` and `!=` are relational operators. You use these operators to determine whether a value is less than or greater than another value of the same type. The following table shows how to use these operators:

Operator	Meaning	Example	Outcome if age is 42
<code>&lt;</code>	Less than	<code>age &lt; 21</code>	false
<code>&lt;=</code>	Less than or equal to	<code>age &lt;= 18</code>	false
<code>&gt;</code>	Greater than	<code>age &gt; 16</code>	true
<code>&gt;=</code>	Greater than or equal to	<code>age &gt;= 42</code>	true

## Understanding conditional logical operators

C# provides two other binary Boolean operators: the logical AND operator, which is represented by the `&&` symbol, and the logical OR operator, which is represented by the `||` symbol. Collectively, these are known as the conditional logical operators. Their purpose is to combine two Boolean expressions or values into a single Boolean result. These operators are similar to the equality and relational operators in that the value of the expressions in which they appear is either `true` or `false`, but they differ in that the values they operate on must also be either `true` or `false`.

The outcome of the `&&` operator is `true` if and only if both of the Boolean expressions it's evaluating are `true`. The following example is taken from the judging criteria for a talent contest. A contestant is assessed on their dancing and singing abilities. A judge awards a value from 0 (no talent whatsoever) to 10 (should be a professional) for each ability. A contestant progresses to the next stage of the contest as long as they score 7 or more for each ability:

```
int dancingAbility = ...; // Value assigned by judge
int singingAbility = ...; // Value assigned by judge
bool canMoveToNextStage;
canMoveToNextStage = (dancingAbility >= 7) && (singingAbility >= 7);
```



**Tip** The parentheses in this example are actually superfluous, but they help clarify the purpose of the expression.

The outcome of the `||` operator is `true` if either of the Boolean expressions it evaluates is `true`. You use the `||` operator to determine whether any one of a combination of Boolean expressions is `true`. For example, the following statement assigns the value `true` to `eliminateContestant` if the value of `dancingAbility` is less than 7 or the value of `singingAbility` is less than 7:

```
bool eliminateContestant;
eliminateContestant = (dancingAbility < 7) || (singingAbility < 7);
```

## Short-circuiting

The `&&` and `||` operators both exhibit a feature called short-circuiting. Short-circuiting describes the situation when it is not necessary to evaluate both operands when ascertaining the result of a conditional logical expression. For example, if the left operand of the `&&` operator evaluates to `false`, the result of the entire expression must be `false`, regardless of the value of the right operand. Similarly, if the value of the left operand of the `||` operator evaluates to `true`, the result of the entire expression must be `true`, irrespective of the value of the right operand. In these cases, the `&&` and `||` operators bypass the evaluation of the right operand. Consider the first example shown previously:

```
(dancingAbility >= 7) && (singingAbility >= 7);
```

In this expression, if the value of `dancingAbility` is less than 7, the Boolean expression on the left side of `&&` evaluates to `false`. This value means that the result of the entire expression must be `false`, and the Boolean expression to the right of the `&&` operator is not evaluated.

Now consider the second example:

```
(dancingAbility < 7) || (singingAbility < 7);
```

Here, if the value of `dancingAbility` is less than 7, the Boolean expression on the left side of `||` evaluates to `true`. This value means that the result of the entire expression must be `true`, and the Boolean expression to the right of the `||` operator is not evaluated.

If you carefully design expressions that use conditional logical operators, you can boost the performance of your code by avoiding unnecessary work. Place simple Boolean expressions that can be evaluated easily on the left side of a conditional logical operator, and put more complex expressions on the right side. In many cases, you'll find that the program does not need to evaluate the more complex expressions.



**Note** A Boolean expression can invoke a Boolean method that returns a `true/false` value. Be aware that in a Boolean expression that uses short-circuiting, the method might not always be run.

## Summarizing operator precedence and associativity

The following table summarizes the precedence and associativity of all the operators you have learned about so far. Operators in the same category have the same precedence. The operators in categories higher up in the table take precedence over operators in categories lower down.

Category	Operators	Description	Associativity
Primary	<code>()</code> <code>++</code> <code>--</code>	Precedence override Post-increment Post-decrement	Left
Unary	<code>!</code> <code>+</code> <code>-</code> <code>++</code> <code>--</code>	Logical NOT Returns the value of the operand unchanged Returns the value of the operand negated Pre-increment Pre-decrement	Left

Category	Operators	Description	Associativity
Multiplicative	* / %	Multiply Divide Division remainder (modulus)	Left
Additive	+ −	Addition Subtraction	Left
Relational	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left
Equality	== !=	Equal to Not equal to	Left
Conditional AND	&&	Conditional AND	Left
Conditional OR		Conditional OR	Left
Assignment	=	Assigns the right-hand operand to the left and returns the value that was assigned	Right

Notice that the `&&` operator and the `||` operator have a different precedence: `&&` is higher than `||`.

## Pattern matching

C# 8.0 includes new pattern-matching capabilities. Patterns extend the decision-making capabilities of many C# statements. They also can make your code more concise and readable than was previously possible.

Pattern matching uses the `is` operator. The easiest way to explain pattern matching for Boolean expressions is through another example. The following statement assigns the value `true` to the Boolean variable `validPercentage` if and only if the value of the `percent` variable is greater than or equal to 0 and the value of `percent` is less than or equal to 100:

```
bool validPercentage;
int percent = ...;
validPercentage = (percent >= 0) && (percent <= 100);
```

Pattern matching enables you to simplify the expression to:

```
validPercentage = (percent is >= 0 and <= 100);
```

The `is` operator works with the Boolean operator `and` (known as the *conjunctive* pattern operator) and the Boolean operator `or` (the *disjunctive* pattern operator). You cannot use `&&` or `||` with `is`. Additionally, pattern matching only matches a single variable against a pattern. For example, the following code, which attempts to evaluate two variables, `dancingAbility` and `singingAbility`, won't work (it doesn't compile):

```
canMoveToNextStage = (dancingAbility is >= 7 and singingAbility is >= 7);
```

C# pattern matching also provides the `not` operator for use with `is`. It is similar to the `!` operator but works only for patterns. The following code shows another way of determining whether the value in the `percent` variable lies within the range 0 to 100:

```
validPercentage = (percent is not < 0 and not > 100);
```



# Using if statements to make decisions

In a method, when you want to choose between executing two different statements depending on the result of a Boolean expression, you can use an `if` statement.

## Understanding if statement syntax

The syntax of an `if` statement is as follows (`if` and `else` are C# keywords):

```
if (booleanExpression)
 statement-1;
else
 statement-2;
```

If `booleanExpression` evaluates to `true`, `statement-1` runs; otherwise, `statement-2` runs. The `else` keyword and the subsequent `statement-2` are optional. If there is no `else` clause and `booleanExpression` evaluates to `false`, execution continues with whatever code follows the `if` statement.



**Note** The Boolean expression must be enclosed in parentheses. Otherwise, the code will not compile.

For example, here's an `if` statement that increments a variable representing the second hand of a stopwatch. (Minutes are ignored for now.) If the value of the `seconds` variable is 59, it is reset to 0; otherwise, it is incremented by using the `++` operator:

```
int seconds;
...
if (seconds == 59)
 seconds = 0;
else
 seconds++;
```

### Boolean expressions only, please!

The expression in an `if` statement must be enclosed in parentheses. Additionally, the expression must be a Boolean expression. In some other languages, notably C and C++, you can write an integer expression, and the compiler will silently convert the integer value to `true` (nonzero) or `false` (0). C# does not support this behavior, and the compiler reports an error if you write such an expression.

If you accidentally specify the assignment operator (`=`) instead of the equality test operator (`==`) in an `if` statement, the C# compiler recognizes your mistake and refuses to compile your code, such as in the following example:

```
int seconds;
...
if (seconds = 59) // compile-time error
...
if (seconds == 59) // ok
```

Accidental assignments were another common source of bugs in C and C++ programs, which would silently convert the value assigned (59) to a Boolean expression (with anything nonzero considered to be true), with the result being that the code following the `if` statement would be performed every time.

Incidentally, you can use a Boolean variable as the expression for an `if` statement, although it must still be enclosed in parentheses, as shown in this example:

```
bool inWord;
...
if (inWord == true) // ok, but not commonly used
...
if (inWord) // more common and considered better style
```

## Using blocks to group statements

Notice that the syntax of the `if` statement shown earlier specifies a single statement after `if ( booleanExpression )` and a single statement after the `else` keyword. Sometimes, however, you'll want to perform more than one statement when a Boolean expression is true.

You could group the statements inside a new method and then call the new method, but a simpler solution is to group the statements inside a block. A block is simply a sequence of statements grouped between an opening brace and a closing brace.

In the following example, two statements that reset the `seconds` variable to 0 and increment the `minutes` variable are grouped inside a block, and the entire block executes if the value of `seconds` is equal to 59:

```
int seconds = 0;
int minutes = 0;
...
if (seconds == 59)
{
 seconds = 0;
 minutes++;
}
else
{
 seconds++;
}
```



**Important** If you omit the braces, the C# compiler associates only the first statement (`seconds = 0;`) with the `if` statement. The subsequent statement (`minutes++;`) will not be recognized by the compiler as part of the `if` statement when the program is compiled. Furthermore, when the compiler reaches the `else` keyword, it will not associate it with the previous `if` statement; instead, it will report a syntax error. Therefore, it is good practice to always define the statements for each branch of an `if` statement within a block, even if a block consists of only a single statement. It might save you some grief later if you want to add more code.

A block also starts a new scope. You can define variables inside a block, but they will disappear at the end of the block. The following code fragment illustrates this point:

```
if (...)
{
 int myVar = 0;
 ... // myVar can be used here
} // myVar disappears here
else
{
 // myVar cannot be used here
 ...
}
// myVar cannot be used here
```

## Cascading if statements

You can nest `if` statements inside other `if` statements. In this way, you can chain together a sequence of Boolean expressions, which are tested one after the other until one of them evaluates to `true`.

In the following example, if the value of `day` is 0, the first test evaluates to `true`, and `dayName` is assigned the string "Sunday". If the value of `day` is not 0, the first test fails, and control passes to the `else` clause, which runs the second `if` statement and compares the value of `day` with 1. The second `if` statement executes only if the first test is false. Similarly, the third `if` statement executes only if the first and second tests are false, and so on.

```
if (day == 0)
{
 dayName = "Sunday";
}
else if (day == 1)
{
 dayName = "Monday";
}
else if (day == 2)
{
 dayName = "Tuesday";
}
else if (day == 3)
{
 dayName = "Wednesday";
}
else if (day == 4)
{
 dayName = "Thursday";
}
else if (day == 5)
{
 dayName = "Friday";
}
else if (day == 6)
{
 dayName = "Saturday";
}
```

```

else
{
 dayName = "unknown";
}

```

In the following exercise, you'll write a method that uses a cascading `if` statement to compare two dates.

### To write `if` statements

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **Selection** solution, which is located in the `\Microsoft Press\VCSBS\Chapter 4\Selection` folder in your **Documents** folder.
3. On the **Debug** menu, select **Start Debugging**.

Visual Studio 2022 builds and runs the application. The form displays two `DatePicker` controls, called `firstDate` and `secondDate`. They are labeled First and Second.

The screenshot shows a window titled "Selection" with a standard Windows toolbar. Below the toolbar, there are two date picker controls. The first is labeled "First:" and shows the date "6 July 2021". The second is labeled "Second:" and also shows the date "6 July 2021". Below these date pickers is a button labeled "Compare". At the bottom of the window is a large, empty rectangular box, likely intended for displaying the result of the comparison.

4. Select **Compare**.

The following text appears in the box in the lower half of the window:

```

firstDate == secondDate : False
firstDate != secondDate : True
firstDate < secondDate : False
firstDate <= secondDate : False
firstDate > secondDate : True
firstDate >= secondDate : True

```

The Boolean expression, `firstDate == secondDate`, should be true because both `firstDate` and `secondDate` are set to the current date. But in fact, only the less-than operator and the greater-than-or-equal-to operator seem to be working correctly.

5. Return to Visual Studio 2022. On the **Debug** menu, select **Stop Debugging**.
6. Display the code for the `MainPage.xaml.cs` file in the Code and Text Editor window.
7. Locate the `compareClick` method, which should look like this:

```
private void compareClick(object sender, RoutedEventArgs e)
{
 int diff = dateCompare(firstDate.Date.LocalDateTime, secondDate.Date.LocalDateTime);
 info.Text = "";
 show("firstDate == secondDate", diff == 0);
 show("firstDate != secondDate", diff != 0);
 show("firstDate < secondDate", diff < 0);
 show("firstDate <= secondDate", diff <= 0);
 show("firstDate > secondDate", diff > 0);
 show("firstDate >= secondDate", diff >= 0);
}
```

This method runs whenever the user selects the Compare button on the form. The expressions `firstDate.Date.LocalDateTime` and `secondDate.Date.LocalDateTime` hold `DateTime` values; they represent the dates displayed in the `firstDate` and `secondDate` controls on the form elsewhere in the application. The `DateTime` data type is just another data type, like `int` or `float`, except it contains sub-elements with which you can access the individual pieces of a date, such as the year, month, or day.

The `compareClick` method passes the two `DateTime` values to the `dateCompare` method. The purpose of this method is to compare dates and return the `int` value 0 if they are the same, -1 if the first date is less than the second, and +1 if the first date is greater than the second. A date is considered greater than another date if it follows it chronologically. You'll examine the `dateCompare` method in the next step.

The `show` method displays the results of the comparison in the info box in the lower half of the form.

8. Locate the `dateCompare` method, which should look like this:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
 // TO DO
 return 42;
}
```

This method currently returns the same value whenever it is called rather than 0, 1, or +1, regardless of the values of its parameters. This explains why the application is not working as expected. You must implement the logic in this method to compare two dates correctly.

9. Remove the `// TO DO` comment and the `return` statement from the `dateCompare` method.

- 10.** Add the following statements shown in bold to the body of the `dateCompare` method:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
 int result = 0;

 if (leftHandSide.Year < rightHandSide.Year)
 {
 result = -1;
 }
 else if (leftHandSide.Year > rightHandSide.Year)
 {
 result = 1;
 }
}
```



**Note** Don't try to build the application yet. The `dateCompare` method is not complete, and the build will fail.

If the expression `leftHandSide.Year < rightHandSide.Year` is true, the date in `leftHandSide` must be earlier than the date in `rightHandSide`, so the program sets the result variable to -1. Otherwise, if the expression `leftHandSide.Year > rightHandSide.Year` is true, the date in `leftHandSide` must be later than the date in `rightHandSide`, and the program sets the result variable to 1.

If the expression `leftHandSide.Year < rightHandSide.Year` is false and the expression `leftHandSide.Year > rightHandSide.Year` is also false, the `Year` property of both dates must be the same, so the program needs to compare the months in each date.

- 11.** Add the following statements shown in bold to the body of the `dateCompare` method. Type them immediately after the code you entered in the preceding step:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
 ...
 else if (leftHandSide.Month < rightHandSide.Month)
 {
 result = -1;
 }
 else if (leftHandSide.Month > rightHandSide.Month)
 {
 result = 1;
 }
}
```

These statements compare months following a logic similar to that used to compare years in the preceding step.

If the expression `leftHandSide.Month < rightHandSide.Month` is false and the expression `leftHandSide.Month > rightHandSide.Month` is also false, the `Month` property of both dates must be the same, so the program finally needs to compare the days in each date.

- 12.** Add the following statements shown in bold to the body of the `dateCompare` method after the code you entered in the preceding two steps. Also, remove the `return 42` statement that you added earlier:

```
private int dateCompare(DateTime leftHandSide, DateTime rightHandSide)
{
 ...
 else if (leftHandSide.Day < rightHandSide.Day)
 {
 result = -1;
 }
 else if (leftHandSide.Day > rightHandSide.Day)
 {
 result = 1;
 }
 else
 {
 result = 0;
 }

 return result;
}
```

You should recognize the form of this logic by now.

If `leftHandSide.Day < rightHandSide.Day` and `leftHandSide.Day > rightHandSide.Day` both are `false`, the value in the `Day` properties in both variables must be the same. The `Month` values and the `Year` values must also be identical, respectively, for the program logic to have reached this point. Therefore, the two dates must be the same, so the program sets the value of `result` to 0.

The final statement returns the value stored in the `result` variable.

- 13.** On the **Debug** menu, select **Start Debugging** to rebuild and run the application.

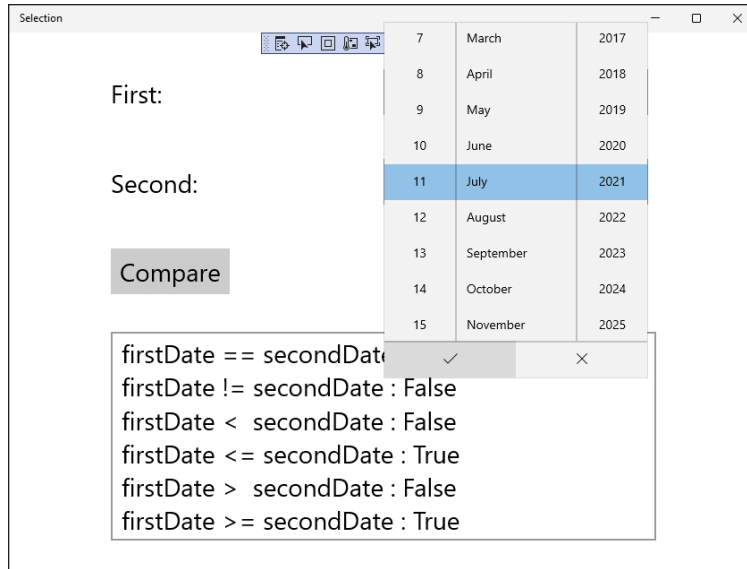
- 14.** Select **Compare**.

The following text appears in the box:

```
firstDate == secondDate : True
firstDate != secondDate : False
firstDate < secondDate: False
firstDate <= secondDate: True
firstDate > secondDate: False
firstDate >= secondDate: True
```

These are the correct results for identical dates.

- 15.** Select the `DatePicker` control to select a later date for the second date (select the tick at the bottom of the control to confirm the selection). Then select **Compare**.



The following text appears in the box:

```

firstDate == secondDate: False
firstDate != secondDate: True
firstDate < secondDate: True
firstDate <= secondDate: True
firstDate > secondDate: False
firstDate >= secondDate: False

```

Again, these are the correct results when the first date is earlier than the second date.

16. Test some other dates and verify that the results are as you would expect.
17. Return to Visual Studio 2022 and stop debugging when you have finished.

## Comparing dates in real-world applications

Now that you have seen how to use a rather long and complicated series of `if` and `else` statements, I should mention that this is not the technique you would employ to compare dates in a real-world application. If you look at the `dateCompare` method from the preceding exercise, you'll see that the two parameters, `leftHandSide` and `rightHandSide`, are `DateTime` values. The logic you have written compares only the date part of these parameters, but they also contain a time element that you have not considered (or displayed). For two `DateTime` values to be considered equal, they should have not only the same date but also the same time. Comparing dates and times is such a common operation that the `DateTime` type actually has a built-in method called `Compare` for doing just that: it compares two `DateTime` arguments and returns a value that indicates whether the first argument is less than the second. If so, the result will be negative; if not, the result will be positive. If both arguments represent the same date and time, the result will be 0.



## Using switch statements

---

Sometimes when you write a cascading `if` statement, each of the `if` statements look similar because they all evaluate an identical expression. The only difference is that each `if` compares the result of the expression with a different value.

For example, consider the following block of code that uses an `if` statement to examine the value in the `day` variable and determine which day of the week it is:

```
if (day == 0)
{
 dayName = "Sunday";
}
else if (day == 1)
{
 dayName = "Monday";
}
else if (day == 2)
{
 dayName = "Tuesday";
}
else if (day == 3)
{
 ...
}
else
{
 dayName = "Unknown";
}
```

Often in these situations, you can rewrite the cascading `if` statement as a `switch` statement to make your program more efficient and more readable.

## Understanding switch statement syntax

The syntax of a `switch` statement is as follows (`switch`, `case`, and `default` are keywords):

```
switch (controllingExpression)
{
 case constantExpression :
 statements
 break;
 case constantExpression :
 statements
 break;
 ...
 default :
 statements
 break;
}
```

The `controllingExpression`, which must be enclosed in parentheses, is evaluated once. Control then jumps to the block of code identified by the `constantExpression`, whose value is equal to the result of the `controllingExpression`. (The `constantExpression` identifier is also called a case label.)

Execution runs as far as the `break` statement, at which point the `switch` statement finishes and the program continues at the first statement that follows the closing brace of the `switch` statement. If none of the `constantExpression` values is equal to the value of the `controllingExpression`, the statements below the optional `default` label run.



**Note** Each `constantExpression` value must be unique so that the `controllingExpression` will match only one of them. If the value of the `controllingExpression` does not match any `constantExpression` value and there is no `default` label, program execution continues with the first statement that follows the closing brace of the `switch` statement.

So, you can rewrite the previous cascading `if` statement as the following `switch` statement:

```
switch (day)
{
 case 0 :
 dayName = "Sunday";
 break;
 case 1 :
 dayName = "Monday";
 break;
 case 2 :
 dayName = "Tuesday";
 break;
 ...
 default :
 dayName = "Unknown";
 break;
}
```

## Following the switch statement rules

The basic `switch` statement is very useful, but any `switch` statement you write must adhere to the following rules:

- You can use `switch` only on certain data types, such as `int`, `char`, or `string`. With any other types, you must use an `if` statement.



**Note** You can use `switch` with `float` and `double` data types, but it may not make sense to do so. This is because floating-point values are held as approximations (albeit very accurate approximations) rather than definitive values. Only numeric types such as `int`, `long`, and `decimal` have guaranteed values.

- The case labels must be constant expressions, such as `42` if the `switch` data type is an `int`, `'42'` if the `switch` data type is a `char`, or `"42"` if the `switch` data type is a `string`. If you need to calculate your case label values at runtime, you must use an `if` statement.

- The case labels must be unique expressions. In other words, two case labels cannot have the same value.
- You can specify that you want to run the same statements for more than one value by providing a list of case labels and no intervening statements. In this case, the code for the final label in the list is executed for all cases in that list. However, if a label has one or more associated statements, execution cannot fall through to subsequent labels; in this case, the compiler generates an error. The following code fragment illustrates these points:

```
switch (trumps)
{
 case "Hearts" :
 case "Diamonds" : // Fall-through allowed - no code between labels
 color = "Red"; // Code executed for Hearts and Diamonds
 break;
 case "Clubs" :
 color = "Black";
 case "Spades" : // Error - code between labels
 color = "Black";
 break;
}
```



**Note** The `break` statement is the most common way to stop fall-through, but you can also use a `return` statement to exit from the method containing the `switch` statement or a `throw` statement to generate an exception and abort the `switch` statement. The `throw` statement is described in Chapter 6, “Managing errors and exceptions.”

### switch fall-through rules

Because you cannot accidentally fall through from one case label to the next if there is any intervening code, you can freely rearrange the sections of a `switch` statement without affecting its meaning (including the `default` label, which by convention is usually, but does not have to be, placed as the last label).

C and C++ programmers should note that the `break` statement is mandatory for every case in a `switch` statement (even the `default` case). This requirement is a good thing because it is common in C or C++ programs to forget the `break` statement, allowing execution to fall through to the next label and leading to bugs that are difficult to spot.

If you really want to, you can mimic C/C++ fall-through in C# by using a `goto` statement to go to the following case or `default` label. Using `goto`, in general, is not recommended, though, and this book does not show you how to do it.

In the following exercise, you’ll complete a program that reads the characters of a string and maps each character to its XML representation. For example, the left angle bracket character (`<`) has a special meaning in XML (it’s used to form elements). If you have data that contains this character, it must be

translated into the text entity `&lt;`; so that an XML processor knows that it is data and not part of an XML instruction. Similar rules apply to the right angle bracket (`>`), ampersand (`&`), single quotation mark (`'`), and double quotation mark (`"`) characters. Here, you'll write a `switch` statement that tests the value of the character and traps the special XML characters as case labels.

### To write switch statements

1. Start Visual Studio 2022 if it is not already running.
2. Open the **SwitchStatement** solution, which is located in the `\Microsoft Press\VCSBS\Chapter 4\SwitchStatement` folder in your **Documents** folder.
3. On the **Debug** menu, select **Start Debugging**.

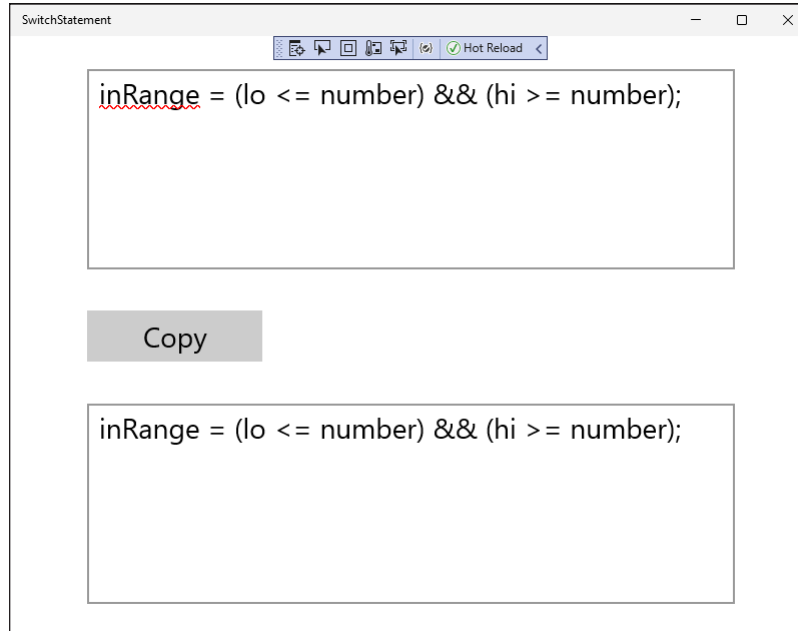
Visual Studio 2022 builds and runs the application. The application displays a form containing two boxes separated by a Copy button.

4. Type the following sample text into the upper box:

```
inRange = (lo <= number) && (hi >= number);
```

5. Select **Copy**.

The statement is copied verbatim into the lower box, and no translation of the `<`, `&`, or `>` characters occurs.



6. Return to Visual Studio 2022 and stop debugging.

7. Display the code for MainPage.xaml.cs in the Code and Text Editor window and locate the copyOne method. It currently looks like this:

```
private void copyOne(char current)
{
 switch (current)
 {
 default:
 target.Text += current;
 break;
 }
}
```

The copyOne method copies the character specified as its input parameter to the end of the text displayed in the lower box. At the moment, copyOne contains a switch statement with a single default action. In the following few steps, you'll modify this switch statement to convert characters that are significant in XML to their XML mapping. For example, the < character will be converted to the string &lt;.

8. Add the following statements shown in bold to the switch statement after the opening brace for the statement and directly before the default label:

```
switch (current)
{
 case '<' :
 target.Text += "<";
 break;
 default:
 target.Text += current;
 break;
}
```

If the current character being copied is a left angle bracket (<), the preceding code appends the string "&lt;" to the text being output in its place.

9. Add the following cases shown in bold to the switch statement, after the break statement you have just added, but above the default label:

```
case '<' :
 target.Text += "<";
 break;
case '>' :
 target.Text += ">";
 break;
case '&' :
 target.Text += "&";
 break;
case '\"' :
 target.Text += """;
 break;
case '\\' :
 target.Text += "'";
 break;
default:
 target.Text += current;
 break;
```



**Note** The single quotation mark (') and double quotation mark (") have a special meaning in C#. They are used to delimit character and string constants. The backslash (\) in the final two case labels is an escape character that causes the C# compiler to treat these characters as literals rather than as delimiters.

10. On the **Debug** menu, select **Start Debugging**.

11. Type the following text into the upper box:

```
inRange = (lo <= number) && (hi >= number);
```

12. Select **Copy**.

The statement is copied into the lower box. This time, each character undergoes the XML mapping implemented in the `switch` statement. The target box displays the following text:

```
inRange = (lo <= number) && (hi >= number);
```

13. Experiment with other strings and verify that all special characters (<, >, &, ", and ') are handled correctly.

14. Return to Visual Studio and stop debugging.

## Using switch expressions with pattern matching

The `switch` statement is a powerful construct but can sometimes be a little verbose and difficult to follow. A `switch` expression is similar to a `switch` statement, except that it yields a value. This sounds like a minor distinction, but it's much more useful than it sounds.

The pattern-matching capabilities of expressions in C# can sometimes make using a `switch` expression a more logical choice than a `switch` statement. Examples include complex validation rules and categorization of data. Suppose you want to determine which range of values a variable named `measurement` falls into. The possible ranges might be `negative` (<0), `zero`, `singledigit` (between 1 and 9), `doubledigit` (between 10 and 99), and `large` (100 or more). The following code shows how you can perform this categorization. The result is stored in the `range` string variable:

```
int measurement = ...;

string range = measurement switch
{
 < 0 => "negative",
 0 => "zero",
 >= 1 and <= 9 => "singledigit",
 >= 10 and <= 99 => "doubledigit",
 >= 100 => "large"
};
```

The syntactic points to note are:

- The `switch` keyword acts as an operator. The measurement variable in the example and the pattern-matching block (the code between the braces) are the operands.
- Each line in the pattern-matching block comprises a pattern and an expression to evaluate as the result. The character sequence `=>` separates the two elements.
- Each pattern/expression pair is separated by a comma. There is no `break` statement.
- The patterns follow the same syntax described for those in `if` statements described earlier.
- The expression to the right of `=>` is evaluated if the pattern matches the first operand of the `switch` operator.
- The `switch` expression must catch every possible value that can be found in the first operand. At runtime, if there is no matching pattern for the operand, you'll receive the error message "The switch expression does not handle all possible values of its input type (it is not exhaustive)." *You can use the underscore character as a catchall that matches any cases not previously handled.*

`switch` expressions also support a range of cases that are not available to `switch` statements. You'll see many examples of these as you proceed through this book.

### To use a switch expression

1. Return to the **SwitchStatement** solution you worked on in the previous exercise.
2. Find the `CopyOne` method. It currently looks like this:

```
private void copyOne(char current)
{
 switch (current)
 {
 case '<':
 target.Text += "<";
 break;
 case '>':
 target.Text += ">";
 break;
 case '&':
 target.Text += "&";
 break;
 case '\"':
 target.Text += """;
 break;
 case '\\'':
 target.Text += "'";
 break;
 default:
 target.Text += current;
 break;
 }
}
```

3. Remove the entire `switch` statement and replace it with the `switch` expression shown in the following code in bold:

```
private void copyOne(char current)
{
 target.Text += current switch
 {
 '<' => "<",
 '>' => ">",
 '&' => "&",
 '\"' => """,
 '\'' => "'",
 _ => current
 };
}
```

This `switch` expression performs pattern matching to determine the value to append to the output displayed on the screen. Notice that the catchall case at the end uses an underscore character (`_`) as the pattern to match. Also, don't forget to add the semicolon to the end of the statement.

This expression is more concise and readable than the `switch` statement used previously.

4. On the **Debug** menu, select **Start Debugging**.
5. Type the following text into the upper box:  
`inRange = (lo <= number) && (hi >= number);`
6. Select **Copy**. Then verify that the program functions exactly as before.
7. Return to Visual Studio and stop debugging.

## Summary

---

In this chapter, you learned about Boolean expressions and variables. You saw how to use Boolean expressions with `if` and `switch` statements to make decisions in your programs, and you combined Boolean expressions by using the Boolean operators.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 5, "Using compound assignment and iteration statements."
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.



## Quick reference

To	Do this
Determine whether two values are equivalent	Use the == operator or the != operator. For example: <code>answer == 42</code>
Compare the value of two expressions	Use the <, <=, >, or >= operator. For example: <code>age &gt;= 21</code>
Declare a Boolean variable	Use the bool keyword as the type of the variable. For example: <code>bool inRange;</code>
Create a Boolean expression that is true only if two conditions are both true	Use the && operator. For example: <code>inRange = (lo &lt;= number) &amp;&amp; (number &lt;= hi);</code>
Create a Boolean expression that is true if either of two conditions is true	Use the    operator. For example: <code>outOfRange = (number &lt; lo)    (hi &lt; number);</code>
Run a statement if a condition is true	Use an if statement. For example: <code>if (inRange)     process();</code>
Run more than one statement if a condition is true	Use an if statement and a block. For example: <code>if (seconds == 59) {     seconds = 0;     minutes++; }</code>
Associate different statements with different values of a controlling expression	Use a switch statement. For example: <code>switch (current) {     case 0:         ...         break;     case 1:         ...         break;     default :         ...         break; }</code>
Assign a value to a variable based on different values of a controlling expression	Use a switch expression. For example: <code>string result = current switch {     &lt; 0 =&gt; "below zero",     0 =&gt; "zero",     &gt; 0 and &lt; 100 =&gt; "between zero and a hundred",     _ =&gt; "greater than a hundred" };</code>

# Using compound assignment and iteration statements

**After completing this chapter, you will be able to:**

- Update the value of a variable by using compound assignment operators.
- Write `while`, `for`, and `do` iteration statements.
- Step through a `do` statement and watch as the values of variables change.

Chapter 4, “Using decision statements,” demonstrated how to use the `if` and `switch` constructs to run statements selectively. In this chapter, you’ll see how to use a variety of iteration (or looping) statements to run one or more statements repeatedly.

When you write iteration statements, you usually need to control the number of iterations you perform. You can achieve this by using a variable, updating its value as each iteration is performed, and stopping the process when the variable reaches a particular value. To help simplify this process, you’ll start by learning about the special assignment operators that you should use to update the value of a variable in these circumstances.

## Using compound assignment operators

You’ve already seen how to use arithmetic operators to create new values. For example, the following statement uses the plus operator (+) to display to the console a value that is 42 greater than the variable `answer`:

```
Console.WriteLine(answer + 42);
```

You’ve also seen how to use assignment statements to change the value of a variable. The following statement uses the assignment operator (=) to change the value of `answer` to 42:

```
answer = 42;
```

If you want to add 42 to the value of a variable, you can combine the assignment operator and the plus operator. For example, the following statement adds 42 to `answer`. After this statement runs, the value of `answer` is 42 more than it was before:

```
answer = answer + 42;
```

Although this statement works, you'll probably never see an experienced programmer write code like this. Adding a value to a variable is so common that C# provides a way for you to perform this task in a shorthand manner by using the operator `+=`. So, for example, to add 42 to `answer`, you can write the following statement:

```
answer += 42;
```

You can use this notation to combine any arithmetic operator with the assignment operator, as the following table shows. These operators are collectively known as compound assignment operators.

Don't write this	Write this
<code>variable = variable * number;</code>	<code>variable *= number;</code>
<code>variable = variable / number;</code>	<code>variable /= number;</code>
<code>variable = variable % number;</code>	<code>variable %= number;</code>
<code>variable = variable + number;</code>	<code>variable += number;</code>
<code>variable = variable - number;</code>	<code>variable -= number;</code>

**Tip** Compound assignment operators share the same precedence and right associativity as the simple assignment operator (`=`).

The `+=` operator also works on strings; it appends one string to the end of another. For example, the following code displays "Hello John" on the console:

```
string name = "John";
string greeting = "Hello ";
greeting += name;
Console.WriteLine(greeting);
```

You cannot use any of the other compound assignment operators on strings.

**Tip** Use the increment (`++`) and decrement (`--`) operators instead of a compound assignment operator when incrementing or decrementing a variable by 1. For example, replace

```
count += 1;

with

count++;
```

## Writing while statements

You use a `while` statement to run a statement repeatedly for as long as some condition is true. The syntax of a `while` statement is as follows:

```
while (booleanExpression)
 statement
```

The Boolean expression (which must be enclosed in parentheses) is evaluated. If it evaluates to `true`, the statement runs, and the Boolean expression is evaluated again. If the expression still evaluates to `true`, the statement is repeated, and the Boolean expression is evaluated yet again. This process continues until the Boolean expression evaluates to `false`, at which point the `while` statement exits. Execution then continues with the first statement that follows the `while` statement.

A `while` statement shares the following syntactic similarities with an `if` statement (in fact, the syntax is identical except for the keyword):

- The expression must be a Boolean expression.
- The Boolean expression must be written within parentheses.
- If the Boolean expression evaluates to `false` when first evaluated, the statement does not run.
- If you want to perform two or more statements under the control of a `while` statement, you must use braces to group those statements in a block.

Here's a `while` statement that writes the values 0 through 9 to the console. Note that as soon as the variable `i` reaches the value 10, the `while` statement finishes and the code in the statement block does not run:

```
int i = 0;
while (i < 10)
{
 Console.WriteLine(i);
 i++;
}
```

All `while` statements should terminate at some point. A common beginner's mistake is to forget to include a statement to cause the Boolean expression eventually to evaluate to `false` and terminate the loop, which results in a program that runs forever. In the example, the statement `i++`; performs this role.



**Note** The variable `i` in the `while` loop controls the number of iterations that the loop performs. This is a common idiom, and the variable that performs this role is sometimes called the sentinel variable. You can also create nested loops (one loop inside another). In these cases, it's common to extend this naming pattern to use the letters `j`, `k`, and even `l` as the names of the sentinel variables used to control the iterations in these loops.



**Tip** As with `if` statements, it's recommended that you always use a block with a `while` statement, even if the block contains only a single statement. This way, if you decide to add more statements to the body of the `while` construct later, it's clear that you should add them to the block. If you don't do this, only the first statement that immediately follows the Boolean expression in the `while` construct will be executed as part of the loop, resulting in difficult-to-spot bugs such as this:

```
int i = 0;
while (i < 10)
 Console.WriteLine(i);
 i++;
```

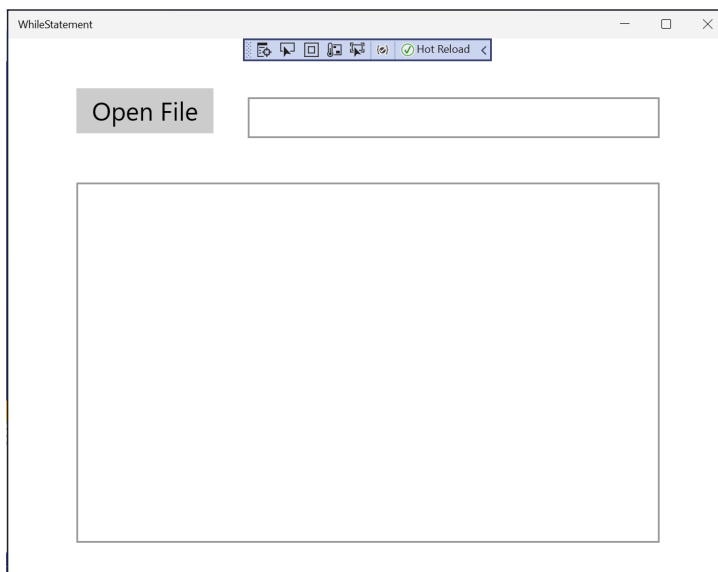
This code iterates forever, displaying an infinite number of zeros, because only the `Console.WriteLine` statement and not the `i++;` statement is executed as part of the `while` construct.

In the following exercise, you'll write a `while` loop to iterate through the contents of a text file one line at a time and write each line to a box in a form.

### To write a while statement

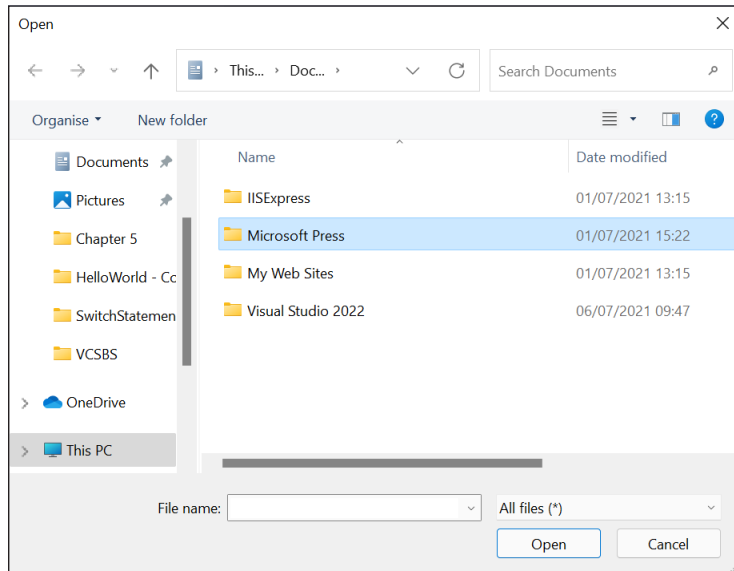
1. Using Microsoft Visual Studio 2022, open the **WhileStatement** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 5\WhileStatement** folder in your **Documents** folder.
2. On the **Debug** menu, select **Start Debugging**.

Visual Studio 2022 builds and runs the application. The application is a simple text file viewer that you can use to select a text file and display its contents.



### 3. Select **Open File**.

The Open dialog opens and displays the files in the Documents folder. (The list of files and folders on your computer might be different from the ones shown here.) You can use this dialog to move to a folder and select a file to display.



4. Move to the **\Microsoft Press\VCSBS\Chapter 5\WhileStatement\WhileStatement** folder in your **Documents** folder.
5. Select the file **MainPage.xaml.cs**, and then select **Open**.

The name of the file, `MainPage.xaml.cs`, appears in the box at the top of the form, but the contents of the file do not appear in the large box. This is because you have not yet implemented the code that reads the contents of the file and displays it. You'll add this functionality in the following steps.

6. Return to Visual Studio 2022 and stop debugging.
7. Display the code for the file `MainPage.xaml.cs` in the Code and Text Editor window and locate the `openFileClick` method.

This method runs when the user selects the Open button to select a file in the Open dialog. It's not necessary for you to understand the exact details of how this method works at this point; simply accept that this method prompts the user for a file (using a `FileOpenPicker` object) and opens the selected file for reading.

The final two statements in the `openFileClick` method are important, however. They look like this:

```
TextReader reader = new StreamReader(inputStream.AsStreamForRead());
displayData(reader);
```

The first statement declares a `TextReader` variable called `reader`. `TextReader` is a class provided by the .NET libraries. You can use this class to read streams of characters from sources such as files. It's located in the `System.IO` namespace. This statement makes the data in the file specified by the user in the `FileOpenPicker` object available to the `TextReader` object, which can then be used to read the data from the file.

The final statement calls a method named `displayData`, passing `reader` as a parameter to this method. The `displayData` method reads the data by using the `reader` object and displays it on-screen (or it will do so once you have written the code to accomplish this task).

8. Examine the `displayData` method. It currently looks like this:

```
private void displayData(TextReader reader)
{
 // TODO: add while loop here

}
```

You can see that, other than the comment, this method is currently empty. This is where you need to add the code to fetch and display the data.

9. Replace the `// TODO: add while loop here` comment with the following statement:

```
source.Text = "";
```

The `source` variable refers to the large box on the form. Setting its `Text` property to the empty string (`""`) clears any text that's currently displayed in this box.

10. Add the following statement after the previous line that you added to the `displayData` method:

```
string line = reader.ReadLine();
```

This statement declares a `string` variable called `line` and calls the `reader.ReadLine` method to read the first line from the file into this variable. This method returns either the next line of text from the file or a special value called `null` when there are no more lines to read.

11. Add the following statements to the `displayData` method after the code you have just entered:

```
while (line is not null)
{
 source.Text += line + '\n';
 line = reader.ReadLine();
}
```

This is a `while` loop that iterates through the file one line at a time until there are no more lines available.

The Boolean expression at the start of the `while` loop examines the value in the `line` variable. If it is not `null`, the body of the loop displays the current line of text by appending it to the `Text` property of the source box, together with a newline character (`'\n'`). (The `ReadLine` method of the `TextReader` object strips out the newline characters as it reads each line, so the code needs to add it back in again.) The `while` loop then reads in the next line of text before performing the next iteration. The `while` loop finishes when there is no more text to read in the file, and the `ReadLine` method returns a `null` value.



**Note** You can also use `!= null` to test for a `null` value. However, the expression `is not null` is more readable.

12. Type the following statement after the closing brace at the end of the `while` loop:

```
reader.Dispose();
```

This statement releases the resources associated with the file and closes it. This is good practice because it makes it possible for other applications to use the file and also frees up any memory and other resources used to access the file.

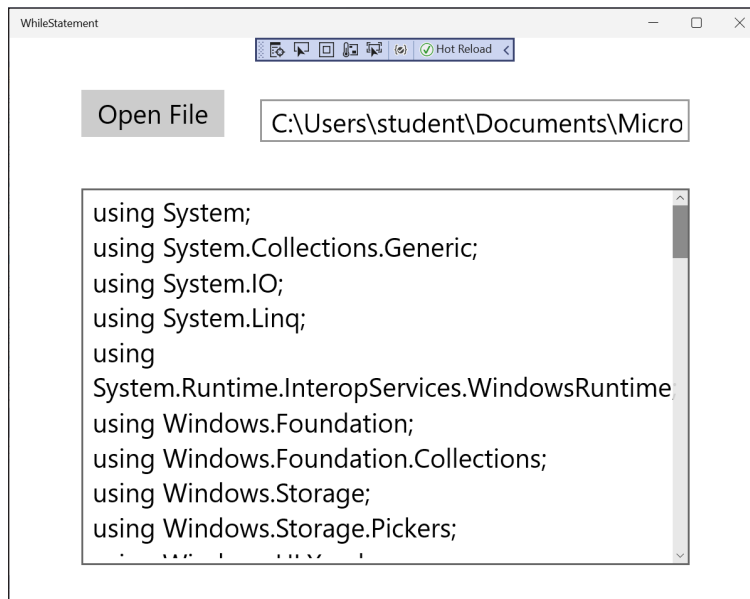
13. On the **Debug** menu, select **Start Debugging**.
14. When the form appears, select **Open File**.
15. In the Open dialog, move to the `\Microsoft Press\VCSBS\Chapter 5\WhileStatement\WhileStatement` folder in your **Documents** folder, select the file `MainPage.xaml.cs`, and then select **Open**.



**Note** Don't try to open a file that does not contain text. If you attempt to open an executable program or a graphics file, for example, the application will simply display a text representation of the binary information in the file. If the file is large, it might hang the application, requiring you to terminate it forcibly.



This time, the contents of the selected file appear in the box; you should recognize the code that you've been editing. The following image shows the application running:



16. Scroll through the text in the box and find the `displayData` method. Verify that this method contains the code you just added.
17. Return to Visual Studio and stop debugging.

## Writing for statements

In C#, most `while` statements have the following general structure:

```
initialization
while (Boolean expression)
{
 statement
 update control variable
}
```

The `for` statement in C# provides a more formal version of this kind of construct by combining the initialization, Boolean expression, and code that updates the control variable. You'll find the `for` statement useful because in a `for` statement, it's much harder to accidentally leave out the code that initializes or updates the control variable, so you are less likely to write code that loops forever. Here's the syntax of a `for` statement:

```
for (initialization; Boolean expression; update control variable) statement
```

The statement that forms the body of the `for` construct can be a single line of code or a code block enclosed in braces.

You can rephrase the `while` loop shown earlier that displays the integers from 0 through 9 as the following `for` loop:

```
for (int i = 0; i < 10; i++)
{
 Console.WriteLine(i);
}
```

The initialization occurs just once, at the very beginning of the loop. Then, if the Boolean expression evaluates to `true`, the statement runs. The control variable update occurs, and then the Boolean expression is reevaluated. If the condition still evaluates to `true`, the statement is executed again, the control variable is updated, the Boolean expression is evaluated again, and so on.

Notice that the initialization occurs only once, that the statement in the body of the loop always executes before the update occurs, and that the update occurs before the Boolean expression reevaluates.



**Tip** As with the `while` statement, it's considered a good practice to always use a code block even if the body of the `for` loop contains just a single statement. If you add more statements to the body of the `for` loop later, this approach will help to ensure that your code is always executed as part of each iteration.

You can omit any of the three parts of a `for` statement. If you omit the Boolean expression, it defaults to `true`, so the following `for` statement runs forever:

```
for (int i = 0; ; i++)
{
 Console.WriteLine("somebody stop me!");
}
```

If you omit the initialization and update parts, you have a strangely spelled `while` loop:

```
int i = 0;
for (; i < 10;)
{
 Console.WriteLine(i);
 i++;
}
```



**Note** The initialization, Boolean expression, and update control variable parts of a `for` statement must always be separated by semicolons, even when they are omitted.

You can also provide multiple initializations and multiple updates in a `for` loop. (You can have only one Boolean expression, though.) To achieve this, separate the various initializations and updates with commas, as shown in the following example:

```
for (int i = 0, j = 10; i <= j; i++, j--)
{
 ...
}
```

As a final example, here's the `while` loop from the preceding exercise recast as a `for` loop:

```
for (string line = reader.ReadLine(); line is not null; line = reader.ReadLine())
{
 source.Text += line + '\n';
}
```

## Understanding for statement scope

You might have noticed that you can declare a variable in the initialization part of a `for` statement. That variable is scoped to the body of the `for` statement and disappears when the `for` statement finishes. This rule has two important consequences. First, you cannot use that variable after the `for` statement has ended because it's no longer in scope. Here's an example:

```
for (int i = 0; i < 10; i++)
{
 ...
}
Console.WriteLine(i); // compile-time error
```

Second, you can write two or more `for` statements that reuse the same variable name because each variable is in a different scope, as shown in the following code:

```
for (int i = 0; i < 10; i++) { ... } for (int i = 0; i < 20; i += 2) // okay { ... }
for (int i = 0; i < 10; i++)
{
 ...
}
for (int i = 0; i < 20; i += 2) // okay
{
 ...
}
```

## Writing do statements

Both the `while` and `for` statements test their Boolean expression at the beginning of the loop. This means that if the expression evaluates to `false` on the first test, the body of the loop does not run (not even once). The `do` statement is different: its Boolean expression is evaluated after each iteration, so the body always executes at least once.

The syntax of the `do` statement is as follows (don't forget the final semicolon):

```
do
 statement
while (booleanExpression);
```

You must use a statement block if the body of the loop contains more than one statement. (The compiler will report a syntax error if you don't.) Here's a version of the example that writes the values 0 through 9 to the console, this time constructed by using a `do` statement:

```
int i = 0;
do
{
 Console.WriteLine(i);
 i++;
}
while (i < 10);
```

## The `break` and `continue` statements

In Chapter 4, you saw how to use the `break` statement to jump out of a `switch` statement. You can also use a `break` statement to jump out of the body of an iteration statement. When you break out of a loop, the loop exits immediately, and execution continues at the first statement that follows the loop. Neither the update nor the continuation condition of the loop is rerun.

In contrast, the `continue` statement causes the program to perform the next iteration of the loop immediately (after reevaluating the Boolean expression). Here's another version that writes the values 0 through 9 to the console, this time using `break` and `continue` statements:

```
int i = 0;
while (true)
{
 Console.WriteLine(i);
 i++;
 if (i < 10)
 continue;
 else
 break;
}
```

This code is absolutely ghastly. Many programming guidelines recommend using `continue` cautiously or not at all because it's often associated with hard-to-understand code. The behavior of `continue` is also quite subtle. For example, if you execute a `continue` statement from within a `for` statement, the update part runs before performing the next iteration of the loop.

In the following exercise, you'll write a `do` statement to convert a positive decimal whole number to its string representation in octal notation. The program is based on the following algorithm, which follows on a well-known mathematical procedure:

```
store the decimal number in the variable dec
do the following
 divide dec by 8 and store the remainder
 set dec to the quotient from the previous step
while dec is not equal to zero
combine the values stored for the remainder for each calculation in reverse order
```

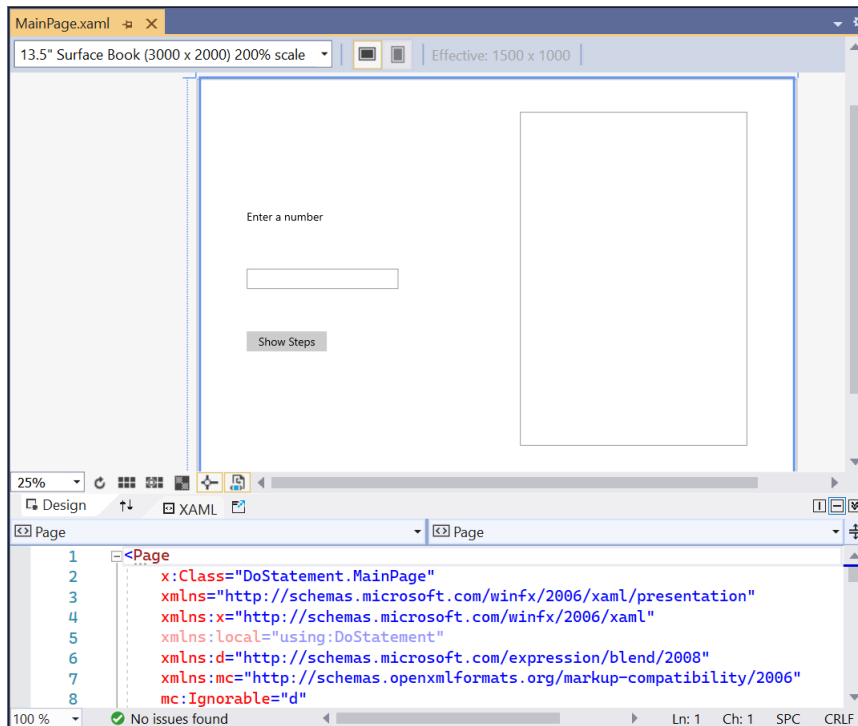
For example, suppose you want to convert the decimal number 999 to octal. You perform the following steps:

1. Divide 999 by 8. The quotient is 124, and the remainder is 7.
2. Divide 124 by 8. The quotient is 15, and the remainder is 4.
3. Divide 15 by 8. The quotient is 1, and the remainder is 7.
4. Divide 1 by 8. The quotient is 0, and the remainder is 1.
5. Combine the values calculated for the remainder at each step in reverse order. The result is 1747. This is the octal representation of the decimal value 999.

### To write a do statement

1. Using Visual Studio 2022, open the **DoStatement** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 5\DoStatement** folder in your **Documents** folder.
2. Display the MainPage.xaml form in the Design View window.

The form contains a box called `number` into which you can enter a decimal number. When you select the Show Steps button, the octal representation of the number entered is generated. The box to the right, called `steps`, shows the results of each stage of the calculation.



3. Display the code for MainPage.xaml.cs in the Code and Text Editor window and locate the showStepsClick method.

This method runs when the user selects the Show Steps button on the form. Currently, the method is empty.

4. Add the following statements shown in bold to the showStepsClick method:

```
private void showStepsClick(object sender, RoutedEventArgs e)
{
 int amount = int.Parse(number.Text);
 steps.Text = "";
 string current = "";
}
```

The first statement converts the string value in the Text property of the number box into an int by using the Parse method of the int type and stores it in a local variable called amount.

The second statement clears the text displayed in the lower box by setting its Text property to the empty string.

The third statement declares a string variable called current and initializes it to the empty string. You'll use this string to store the digits generated at each iteration of the loop that's used to convert the decimal number to its octal representation.

5. Add the following do statement (shown in bold) to the showStepsClick method:

```
private void showStepsClick(object sender, RoutedEventArgs e)
{
 int amount = int.Parse(number.Text);
 steps.Text = "";
 string current = "";

 do
 {
 int nextDigit = amount % 8;
 amount /= 8;
 int digitCode = '0' + nextDigit;
 char digit = Convert.ToChar(digitCode);
 current = digit + current;
 steps.Text += current + "\n";
 }
 while (amount != 0);
}
```

The algorithm used here repeatedly performs integer arithmetic to divide the amount variable by 8 and determine the remainder. The remainder after each successive division constitutes the next digit in the string being built. Eventually, when amount is reduced to 0, the loop finishes.

Notice that the body must run at least once. This behavior is exactly what is required because even the number 0 has one octal digit.

Look more closely at the code; you'll see that the first statement executed by the do loop is this:

```
int nextDigit = amount % 8;
```

This statement declares an `int` variable called `nextDigit` and initializes it to the remainder after dividing the value in `amount` by 8. This will be a number somewhere between 0 and 7.

The next statement in the do loop is:

```
amount /= 8;
```

This is a compound assignment statement and is equivalent to writing `amount = amount / 8;`. If the value of `amount` is 999, the value of `amount` after this statement runs is 124.

The next statement is this:

```
int digitCode = '0' + nextDigit;
```

This statement requires a little explanation. Characters have a unique code according to the character set used by the operating system. In the character sets frequently used by the Windows operating system, the code for character '0' has the integer value 48. The code for character '1' is 49, the code for character '2' is 50, and so on, up to the code for character '9', which has the integer value 57.

With C#, you can treat a character as an integer and perform arithmetic on it, but when you do so, C# uses the character's code as the value. So, the expression `'0' + nextDigit` actually results in a value somewhere between 48 and 55 (remember that `nextDigit` will be between 0 and 7), corresponding to the code for the equivalent octal digit.

The fourth statement in the do loop is:

```
char digit = Convert.ToChar(digitCode);
```

This statement declares a `char` variable called `digit` and initializes it to the result of the `Convert.ToChar(digitCode)` method call. The `Convert.ToChar` method takes an integer holding a character code and returns the corresponding character. So, for example, if `digitCode` has the value 54, `Convert.ToChar(digitCode)` returns the character '6'.

To summarize, the first four statements in the do loop have determined the character representing the least-significant (rightmost) octal digit corresponding to the number the user entered. The next task is to prepend this digit to the string to be output, like this:

```
current = digit + current;
```

The next statement in the do loop is this:

```
steps.Text += current + "\n";
```

This statement adds to the Steps box the string containing the digits produced so far for the octal representation of the number. It also appends a newline character so that each stage of the conversion appears on a separate line in the box.

Finally, the condition in the `while` clause at the end of the loop is evaluated:

```
while (amount != 0);
```

Because the value of `amount` is not yet 0, the loop performs another iteration.

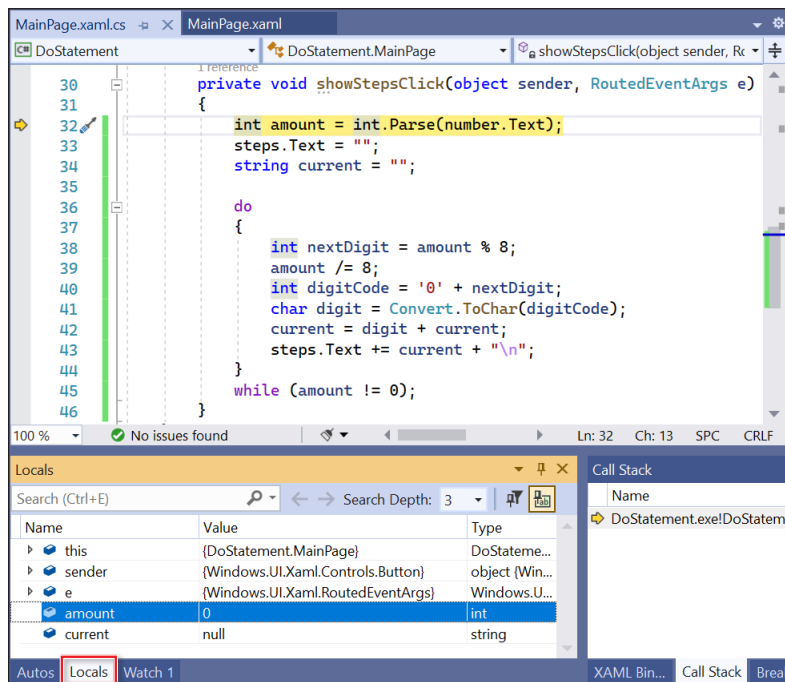
In the final exercise of this chapter, you'll use the Visual Studio 2022 debugger to step through the previous do statement to help you understand how it works.

### To step through the do statement

1. In the Code and Text Editor window displaying the MainPage.xaml.cs file, move the cursor to the first statement of the showStepsClick method:
2. Right-click anywhere in the first statement, and then select **Run To Cursor**.
3. When the form appears, type **999** in the Number box on the left, and then select **Show Steps**.

The program stops, and you are placed in Visual Studio 2022 Debug mode. A yellow arrow in the left margin of the Code and Text Editor window and yellow highlighting on the code indicates the current statement.

4. In the window below the Code and Text Editor window, select the **Locals** tab, as highlighted in the following image:



The Locals window displays the name, value, and type of the local variables in the current method, including the `amount` local variable. Notice that the value of `amount` is currently 0.

5. Display the Debug toolbar if it is not visible. (On the **View** menu, point to **Toolbars**, and then select **Debug**.)





**Note** The commands on the Debug toolbar are also available on the Debug menu displayed on the menu bar.

6. On the Debug toolbar, select the **Step Into** button.

The debugger runs the following statement:

```
int amount = int.Parse(number.Text);
```

The value of `amount` in the Locals window changes to 999, and the yellow arrow moves to the next statement.

7. Select **Step Into** again.

The debugger runs this statement:

```
steps.Text = "";
```

This statement does not affect the Locals window because `steps` is a control on the form and not a local variable. The yellow arrow moves to the next statement.

8. Select **Step Into**.

The debugger runs the statement shown here:

```
string current = "";
```

The yellow arrow moves to the opening brace at the start of the do loop. The do loop contains three local variables of its own: `nextDigit`, `digitCode`, and `digit`. Notice that these local variables now appear in the Locals window. The value of all three variables is initially set to 0.

9. Select **Step Into**.

The yellow arrow moves to the first statement within the do loop.

10. Select **Step Into**.

The debugger runs the following statement:

```
int nextDigit = amount % 8;
```

The value of `nextDigit` in the Locals window changes to 7. This is the remainder after dividing 999 by 8.

11. Select **Step Into**.

The debugger runs this statement:

```
amount /= 8;
```

The value of `amount` changes to 124 in the Locals window.

## 12. Select **Step Into**.

The debugger runs this statement:

```
int digitCode = '0' + nextDigit;
```

The value of `digitCode` in the Locals window changes to 55. This is the character code of the character '7' (48 + 7).

## 13. Select **Step Into**.

The debugger continues to this statement:

```
char digit = Convert.ToChar(digitCode);
```

The value of `digit` changes to '7' in the Locals window. The Locals window shows `char` values using both the underlying numeric value (in this case, 55) and also the character representation ('7').

Note that in the Locals window, the value of the current variable is still "".

## 14. Select **Step Into**.

The debugger runs the following statement:

```
current = current + digit;
```

The value of `current` changes to '7' in the Locals window.

## 15. Select **Step Into**.

The debugger runs the statement shown here:

```
steps.Text += current + "\n";
```

This statement displays the text 7 in the Steps box, followed by a newline character to cause subsequent output to be displayed on the next line in the box. (The form is currently hidden behind Visual Studio, so you won't be able to see it.) The cursor moves to the closing brace at the end of the `do` loop.

## 16. Select **Step Into**.

The yellow arrow moves to the `while` statement to evaluate whether the `do` loop has completed or whether it should continue for another iteration.

## 17. Select **Step Into**.

The debugger runs this statement:

```
while (amount != 0);
```

The value of `amount` is 124, and the expression `124 != 0` evaluates to `true`, so the `do` loop performs another iteration. The yellow arrow jumps back to the opening brace at the start of the `do` loop.

**18. Select **Step Into**.**

The yellow arrow moves to the first statement within the do loop again.

**19. Repeatedly select **Step Into** to step through the next three iterations of the do loop and watch how the values of the variables change in the Locals window.**

**20. At the end of the fourth iteration of the loop, the value of `amount` is 0, and the value of `current` is "1747". The yellow arrow is on the `while` condition at the end of the do loop:**

```
while (amount != 0);
```

Because the value of `amount` is now 0, the expression `amount != 0` evaluates to `false`, and the do loop terminates.

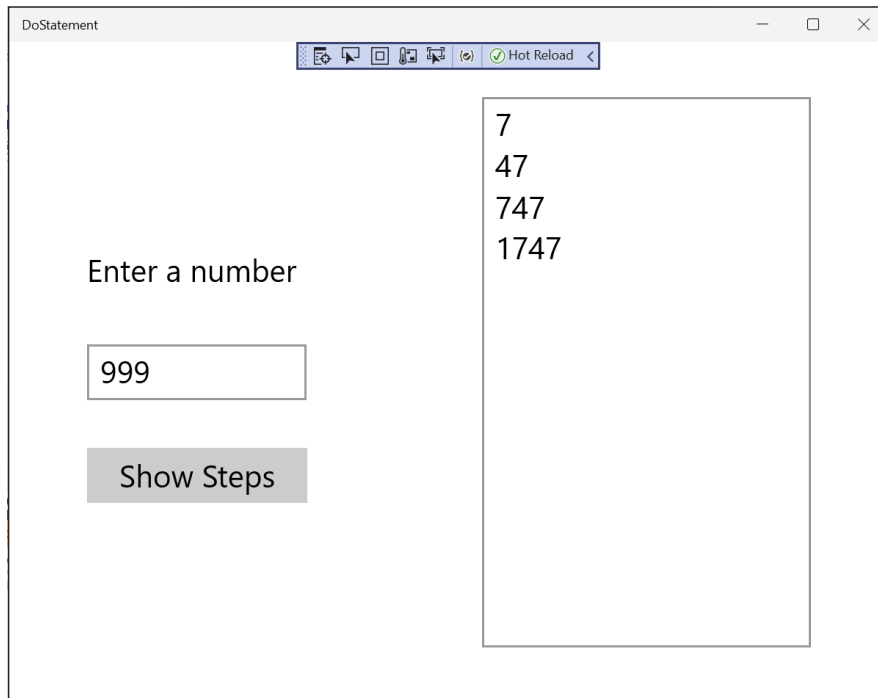
**21. Select **Step Into**.**

The debugger evaluates the `while` statement.

As predicted, the do loop finishes, and the yellow arrow moves to the closing brace at the end of the `showStepsClick` method.

**22. On the **Debug** menu, select **Continue**.**

The form appears, displaying the four steps used to create the octal representation of 999: 7, 47, 747, and 1747.



**23. Return to Visual Studio 2022. On the **Debug** menu, select **Stop Debugging**.**

# Summary

In this chapter, you learned how to use compound assignment operators to update numeric variables and append one string to another. You saw how to use `while`, `for`, and `do` statements to execute code repeatedly while some Boolean condition is `true`.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 6, “Managing errors and exceptions.”
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

# Quick reference

To	Do this
Add an amount to a variable	Use the compound addition operator. For example: <code>variable += amount;</code>
Subtract an amount from a variable	Use the compound subtraction operator. For example: <code>variable -= amount;</code>
Run one or more statements zero or more times while a condition is true	Use a <code>while</code> statement. For example: <pre>int i = 0; while (i &lt; 10) {     Console.WriteLine(i);     i++; }</pre> Alternatively, use a <code>for</code> statement. For example: <pre>for (int i = 0; i &lt; 10; i++) {     Console.WriteLine(i); }</pre>
Repeatedly execute statements one or more times	Use a <code>do</code> statement. For example: <pre>int i = 0; do {     Console.WriteLine(i);     i++; } while (i &lt; 10);</pre>



# Managing errors and exceptions

**After completing this chapter, you will be able to:**

- Handle exceptions by using the `try`, `catch`, and `finally` statements.
- Control integer overflow by using the `checked` and `unchecked` keywords.
- Raise exceptions from your own methods by using the `throw` keyword.
- Ensure that code always runs, even after an exception has occurred, by using a `finally` block.

You have now seen the core C# statements that you need to know to perform common tasks such as writing methods, declaring variables, using operators to create values, writing `if` and `switch` statements to run code selectively, and writing `while`, `for`, and `do` statements to run code repeatedly. However, the previous chapters haven't considered the possibility (or probability) that things can go wrong.

It's a fact of life that bad things sometimes happen. Tires are punctured, batteries run down, screwdrivers are never where you left them, and users of your applications behave in unpredictable ways. In the world of computers, hard disks become corrupt, other applications running on the same computer as your program run amok and use up all the available memory, and wireless network connections disappear at the most awkward moments. Even natural phenomena such as a nearby lightning strike can have an impact if it causes a power outage or network failure. Errors can occur at almost any stage when a program runs, and many errors might not actually be the fault of your application. So how do you detect them and attempt to recover?

Over the years, several mechanisms have evolved. A typical approach adopted by older systems such as UNIX involved arranging for the operating system to set a special global variable whenever a method failed. Then, after each call to a method, you checked the global variable to see whether the method succeeded. C# and most other modern object-oriented languages don't handle errors in this manner; it's just too painful. Instead, they use exceptions. If you want to write robust C# programs, you need to know about exceptions.

In this final chapter of Part I, you'll learn how C# uses exceptions to signal that an error has occurred and how to use the `try`, `catch`, and `finally` statements to catch and handle the errors that these exceptions represent. By the end of this chapter, you'll have a solid foundation in all the fundamental elements of C#, and you'll build on this foundation in Part II.

## Trying code and catching exceptions

---

It's very difficult to ensure that a piece of code always works as expected. Failures can occur for numerous reasons, many of which are beyond your control as a programmer. Any applications that you write must be capable of detecting failures and gracefully handling them, either by taking the appropriate corrective actions or, if that's not possible, by reporting the reasons for the failure in the clearest possible way to the user.

Errors can happen at any time. Using traditional techniques to manually add error-detecting code around every statement is cumbersome, time-consuming, and itself error-prone. You can also lose sight of the main flow of an application if each statement requires contorted error-handling logic to manage each possible error that can occur at every stage.

Fortunately, C# makes it easy to separate the error-handling code from the code that implements the primary logic of a program by using exceptions and exception handlers. To write exception-aware programs, you need to do two things:

- Write your code within a `try` block (`try` is a C# keyword). When the code runs, it attempts to execute all the statements in the `try` block, and if none of the statements generates an exception, they all run, one after the other, to completion. However, if an error condition occurs, execution jumps out of the `try` block and into another piece of code designed to catch and handle the exception: a catch handler.
- Write one or more catch handlers (`catch` is another C# keyword) immediately after the `try` block to handle any possible error conditions. A catch handler is intended to capture and handle a specific type of exception. You can have multiple catch handlers after a `try` block, each one designed to trap and process a specific exception. This enables you to provide different handlers for the different errors that could arise in the `try` block. If any one of the statements within the `try` block causes an error, the runtime throws an exception. The runtime then examines the catch handlers after the `try` block and transfers control directly to the first matching handler.

Here's an example of a `try` block that contains code that attempts to convert strings that a user has typed in some boxes on a form to integer values. The code then calls a method to calculate a value and writes the result to another box. Converting a string to an integer requires that the string contain a valid set of digits and not some arbitrary sequence of characters. If the string contains invalid characters, the `int.Parse` method throws a `FormatException` and execution transfers to the corresponding catch handler. When the catch handler finishes, the program continues with the first statement that follows the handler. (If no handler corresponds to the exception, the exception is said to be unhandled; this situation will be described shortly.)

```
try
{
 int leftHandSide = int.Parse(lhsOperand.Text);
 int rightHandSide = int.Parse(rhsOperand.Text);
 int answer = doCalculation(leftHandSide, rightHandSide);
 result.Text = answer.ToString();
}
```

```

catch (FormatException fEx)
{
 // Handle the exception
 ...
}

```

A catch handler employs syntax similar to that used by a method parameter to specify the exception to be caught. In the preceding example, when a `FormatException` is thrown, the `fEx` variable is populated with an object containing the details of the exception.

The `FormatException` type has several properties that you can examine to determine the exact cause of the exception. Many of these properties are common to all exceptions. For example, the `Message` property contains a text description of the error that caused the exception. You can use this information when handling the exception, perhaps recording the details in a log file or displaying a meaningful message to the user and then asking the user to try again.

## Unhandled exceptions

What happens if a try block throws an exception and there's no corresponding catch handler? In the previous example, it's possible that the `lhsOperand` box could contain the string representation of a valid integer but the integer it represents is outside the range of valid integers supported by C# (for example, "2147483648"). In this case, the `int.Parse` statement would throw an `OverflowException`, which will not be caught by the `FormatException` catch handler. If this occurs and the try block is part of a method, the method immediately exits and execution returns to the calling method. If the calling method uses a try block, the runtime attempts to locate and execute a matching catch handler for this try block. If the calling method does not use a try block or if there is no matching catch handler, the calling method immediately exits, and execution returns to its caller, where the process is repeated. If a matching catch handler is eventually found, the handler runs and execution continues with the first statement that follows the catch handler in the catching method.

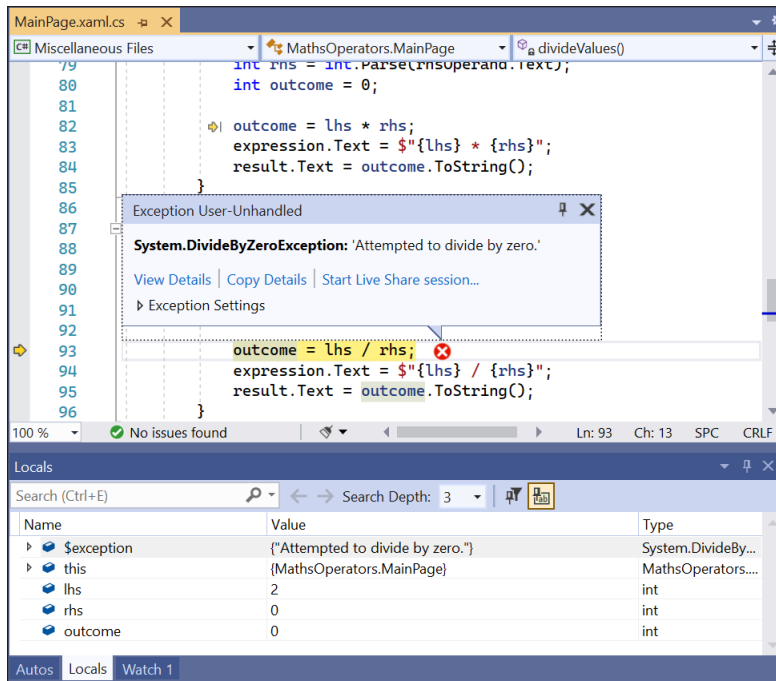


**Important** Notice that after catching an exception, execution continues in the method containing the catch block that caught the exception. If the exception occurred in a method other than the one containing the catch handler, control does not return to the method that caused the exception.

If, after cascading back through the list of calling methods, the runtime is unable to find a matching catch handler, the program will terminate with an unhandled exception.

You can easily examine exceptions generated by your application. If you're running the application in Microsoft Visual Studio 2022 in debug mode (that is, on the Debug menu, you selected Start Debugging to run the application) and an exception occurs, a dialog similar to the one shown here appears, and the application pauses, helping you determine the cause of the exception.





The application stops at the statement that caused the exception and drops you into the debugger. There you can examine the values of variables, change the values of variables, and step through your code from the point at which the exception occurred by using the Debug toolbar and the various debug windows.

## Using multiple catch handlers

The previous discussion highlighted how different errors throw different kinds of exceptions to represent different kinds of failures. To cope with these situations, you can supply multiple catch handlers, one after the other, such as in the following:

```
try
{
 int leftHandSide = int.Parse(lhsOperand.Text);
 int rightHandSide = int.Parse(rhsOperand.Text);
 int answer = doCalculation(leftHandSide, rightHandSide);
 result.Text = answer.ToString();
}
catch (FormatException fEx)
{
 //..
}
catch (OverflowException oEx)
{
 //...
}
```

If the code in the `try` block throws a `FormatException` exception, the statements in the `catch` block for the `FormatException` exception run. If the code throws an `OverflowException` exception, the `catch` block for the `OverflowException` exception runs.



**Note** If the code in the `FormatException` catch block generates an `OverflowException` exception, it does not cause the adjacent `OverflowException` catch block to run. Instead, the exception propagates to the method that invoked this code, as described earlier in this section.

## Catching multiple exceptions

The exception-catching mechanism provided by C# and .NET is quite comprehensive. The .NET libraries define many types of exceptions, and any programs you write can throw most of them.

It's highly unlikely that you'll want to write catch handlers for every possible exception that your code can throw. Remember, your application must be able to handle exceptions that you never even considered when you wrote it! So, how do you ensure that your programs catch and handle all possible exceptions?

The answer to this question lies in the way the different exceptions are related to one another. Exceptions are organized into families called inheritance hierarchies. (You'll learn about inheritance in Chapter 12, "Working with inheritance.") `FormatException` and `OverflowException` both belong to a family called `SystemException`, as do a number of other exceptions. `SystemException` is itself a member of a wider family simply called `Exception`, and this is the great-granddaddy of all exceptions. If you catch `Exception`, the handler traps every possible exception that can occur.



**Note** The `Exception` family includes a wide variety of exceptions, many of which are intended for use by various parts of .NET. Some of these exceptions are somewhat esoteric, but it's still useful to understand how to catch them.

The next example shows how to catch all possible exceptions:

```
try
{
 int leftHandSide = int.Parse(lhsOperand.Text);
 int rightHandSide = int.Parse(rhsOperand.Text);
 int answer = doCalculation(leftHandSide, rightHandSide);
 result.Text = answer.ToString();
}
catch (Exception ex) // this is a general catch handler
{
 //...
}
```



**Note** If you want to catch `Exception`, you can actually omit its name from the catch handler because it is the default exception:

```
catch
{
 // ...
}
```

However, this is not recommended. The exception object passed to the catch handler can contain useful information concerning the exception, which is not easily accessible when using this version of the catch construct.

There's one final question you should be asking at this point: What happens if the same exception matches multiple catch handlers at the end of a try block? For example, if you catch `FormatException` and `Exception` in two different handlers, which one will run? (Or will both execute?)

When an exception occurs, the runtime uses the first handler it finds that matches the exception, and the others are ignored. So, if you place a handler for `Exception` before a handler for `FormatException`, the `FormatException` handler will never run. Therefore, you should place more specific catch handlers above a general catch handler after a try block. If none of the specific catch handlers matches the exception, the general catch handler will.

## Filtering exceptions

You can filter the exceptions that are matched against catch handlers to ensure that a catch handler is triggered only when additional conditions are met. These conditions take the form of a Boolean expression prefixed by the `when` keyword. The following example illustrates the syntax:

```
bool catchErrors = ...;
try
{
 ...
}
catch (Exception ex) when (catchErrors == true)
{
 // Handle exceptions only if the catchErrors variable is true
}
```

This example catches all exceptions (the `Exception` type) depending on the value of the `catchErrors` Boolean variable. If this variable is `false`, then no exception handling occurs, and the default exception handling mechanism for the application is used. If `catchErrors` is `true`, then the code in the catch block runs to handle the exception.

In the following exercises, you'll see what happens when an application throws an unhandled exception, and then you'll write a try block to catch and handle an exception.

## To observe how the application reports unhandled exceptions

1. Start Visual Studio 2022, if it is not already running.
2. Open the **MathOperators** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 6\MathOperators** folder in your **Documents** folder.

This is a version of the program from Chapter 2, “Working with variables, operators, and expressions,” that demonstrates the different arithmetic operators.

3. On the **Debug** menu, select **Start Without Debugging**.



**Note** For this exercise, ensure that you actually run the application *without* debugging.

The form appears. You will now enter some text in the Left Operand box that will cause an exception. This operation will demonstrate the lack of robustness in this current version of the program.

4. In the **Left Operand** box, type **John**, and in the **Right Operand** box, type **2**.
5. Select the **+ Addition** button, and then select **Calculate**.

This input triggers Windows default exception handling. The application simply terminates, and you are returned to the desktop without any explanation of what went wrong!

Now that you have seen how the application behaves when an unhandled exception occurs, the next step is to make the application more robust by handling invalid input and preventing unhandled exceptions from arising.

## To write a try/catch statement block

1. Return to Visual Studio 2022.
2. On the **Debug** menu, select **Start Debugging**.



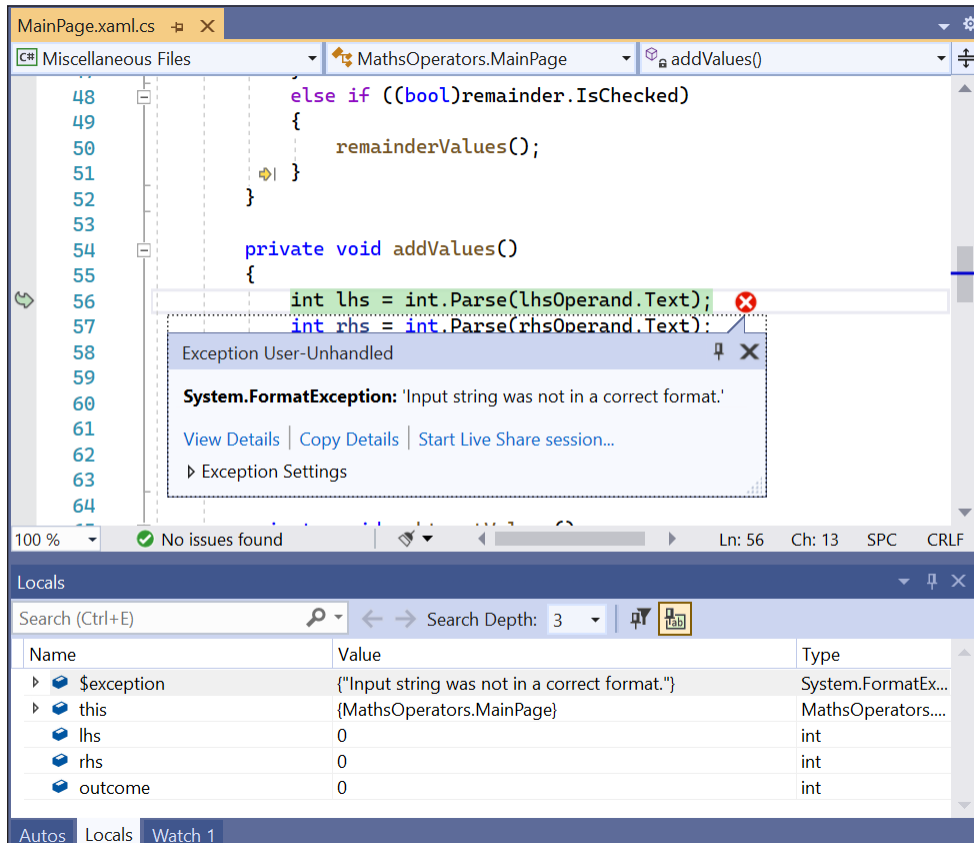
**Note** This time, make sure you run the application in debug mode.

3. When the form appears, in the **Left Operand** box, type **John**, and in the **Right Operand** box, type **2**.

4. Select the + **Addition** button, and then select **Calculate**.

This input should cause the same exception that occurred in the previous exercise, except now you're running in debug mode, so Visual Studio traps the exception and reports it.

Visual Studio displays your code and highlights the statement that caused the exception. It also displays a dialog that describes the exception, which in this case is "Input string was not in a correct format."



You can see that a `FormatException` was thrown by the call to `int.Parse` inside the `addValues` method. The problem is that this method is unable to parse the text `John` into a valid number.



9. Add a try block (including braces) around the statements inside this method, together with a catch handler for the `FormatException` exception, as shown in bold here:

```
try
{
 int lhs = int.Parse(lhsOperand.Text);
 int rhs = int.Parse(rhsOperand.Text);
 int outcome = 0;

 outcome = lhs + rhs;
 expression.Text = $"{lhs} + {rhs}";
 result.Text = outcome.ToString();
}
catch (FormatException fEx)
{
 result.Text = fEx.Message;
}
```

Now, if a `FormatException` exception occurs, the catch handler displays the text held in the exception's `Message` property in the Result box at the bottom of the form.

10. On the **Debug** menu, select **Start Debugging**.
11. When the form appears, in the **Left Operand** box, type **John**, and in the **Right Operand** box type **2**.
12. Select the **+ Addition** button, and then select **Calculate**.

The catch handler successfully catches the `FormatException`, and the message "Input string was not in a correct format" is written to the Result box. The application is now a bit more robust.

The screenshot shows a Windows application window titled "MathsOperators". The window contains a form with the following elements:

- Left Operand:** A text box containing the text "John".
- Right Operand:** A text box containing the text "2".
- Operators:** A list of radio buttons for mathematical operations:
  - ☒ + Addition
  - ☐ - Subtraction
  - ☐ \* Multiplication
  - ☐ / Division
  - ☐ % Remainder
- Calculate:** A button with a grey background.
- Expression:** A label below the Calculate button.
- Result:** A label followed by a text box containing the error message: "Input string was not in a correct format."

The application is running in a debug environment, as indicated by the toolbar at the top which includes icons for debugging and a "Hot Reload" button.

13. In the **Left Operand** box, replace John with the number **10**. In the **Right Operand** box, type **Sharp**. Then select **Calculate**.

The `try` block surrounds the statements that parse both boxes, so the same exception handler handles user input errors in both boxes.

14. In the **Right Operand** box, replace Sharp with **20**, select the **+ Addition** button, and then select **Calculate**.

The application now works as expected and displays the value 30 in the Result box.

15. In the **Left Operand** box, replace 10 with **John**, select the **- Subtraction** button, and then select **Calculate**.

Visual Studio drops into the debugger and reports a `FormatException` exception again. This time, the error has occurred in the `subtractValues` method, which does not include the necessary `try/catch` processing. You'll fix this problem shortly.

16. On the **Debug** menu, select **Stop Debugging**.

## Propagating exceptions

Adding a `try/catch` block to the `addValues` method has made that method more robust, but you need to apply the same exception handling to the other methods: `subtractValues`, `multiplyValues`, `divideValues`, and `remainderValues`. The code for each of these exception handlers will likely be very similar, resulting in you writing the same code in each method.

Each of these methods is called by the `calculateClick` method when the user selects the Calculate button. Therefore, to avoid duplication of the exception-handling code, it makes sense to relocate it to the `calculateClick` method. If a `FormatException` occurs in the `subtractValues`, `multiplyValues`, `divideValues`, or `remainderValues` method, it will be propagated back to the `calculateClick` method for handling as described in the section "Unhandled exceptions" earlier in this chapter.

### To propagate an exception back to the calling method

1. Display the code for the file `MainPage.xaml.cs` in the Code and Text Editor window, and locate the `addValues` method.
2. Remove the `try` block and catch handler from the `addValues` method and return it to its original state, as shown in the following code:

```
private void addValues()
{
 int leftHandSide = int.Parse(lhsOperand.Text);
 int rightHandSide = int.Parse(rhsOperand.Text);
 int outcome = 0;

 outcome = lhs + rhs;
 expression.Text = lhsOperand.Text + " + " + rhsOperand.Text
 result.Text = outcome.ToString();
}
```



3. Find the `calculateClick` method. Add to this method the `try` block and catch handler shown in bold in the following example:

```
private void calculateClick(object sender, RoutedEventArgs e)
{
 try
 {
 if ((bool)addition.IsChecked)
 {
 addValues();
 }
 else if ((bool)subtraction.IsChecked)
 {
 subtractValues();
 }
 else if ((bool)multiplication.IsChecked)
 {
 multiplyValues();
 }
 else if ((bool)division.IsChecked)
 {
 divideValues();
 }
 else if ((bool)remainder.IsChecked)
 {
 remainderValues();
 }
 }
 catch (FormatException fEx)
 {
 result.Text = fEx.Message;
 }
}
```

4. On the **Debug** menu, select **Start Debugging**.
5. When the form appears, in the **Left Operand** box, type **John**, and in the **Right Operand** box, type **2**.
6. Select the **+ Addition** button, and then select **Calculate**.

As before, the catch handler successfully catches the `FormatException`, and the message "Input string was not in a correct format" is written to the Result box. However, although the exception was thrown in the `addValues` method, the handler caught it in the `calculateClick` method.

7. Select the **- Subtraction** button, and then select **Calculate**.

This time, the `subtractValues` method causes the exception, but it is propagated back to the `calculateClick` method and handled in the same manner as before.

8. Select the **\* Multiplication**, **/ Division**, and **% Remainder** buttons and verify that the `FormatException` exception is caught and handled correctly.
9. Return to Visual Studio and stop debugging.



**Note** The decision as to whether to catch unhandled exceptions explicitly in a method depends on the nature of the application you're building. In some cases, it makes sense to catch exceptions as close as possible to the point at which they occur. In other situations, it's more useful to let an exception propagate back to the method that invoked the routine that threw the exception and handle the error there.

## Using checked and unchecked integer arithmetic

Chapter 2 discussed how to use binary arithmetic operators such as `+` and `*` on primitive data types such as `int` and `double`. You should also recall that the primitive data types have a fixed size. For example, a C# `int` is 32 bits. Because `int` has a fixed size, you know exactly the range of values that it can hold: `-2147483648` to `2147483647`.



**Tip** If you want to refer to the minimum or maximum value of `int` in code, you can use the `int.MinValue` or `int.MaxValue` properties.

The fixed size of the `int` type creates a problem. For example, what happens if you add 1 to an `int` whose value is currently `2147483647`? The answer is that it depends on how the application is compiled. By default, the C# compiler generates code that allows the calculation to overflow silently, and you get the wrong answer. (In fact, the calculation wraps around to the largest negative integer value, and the result generated is `-2147483648`.) The reason for this behavior is performance: integer arithmetic is a common operation in almost every program, and adding the overhead of overflow checking to each integer expression could lead to very poor performance. In many cases, the risk is acceptable because you know (or hope!) that your `int` values won't reach their limits. If you don't like this approach, you can turn on overflow checking.



**Tip** You turn overflow checking on and off in Visual Studio 2022 by setting the project properties. In Solution Explorer, select `YourProject` (where `YourProject` is the actual name of the project). On the Project menu, select `YourProject Properties`. In the Project Properties dialog, select the Build tab. Select the Advanced button in the lower-right corner of the page. Finally, in the Advanced Build Settings dialog, select or clear the Check For Arithmetic Overflow/Underflow checkbox.

Regardless of how you compile an application, you can use the `checked` and `unchecked` keywords to turn on and off integer arithmetic overflow checking in parts of an application that you think need it (or don't). These keywords override the compiler option specified for the project.

## Writing checked statements

A checked statement is a block preceded by the checked keyword. All integer arithmetic in a checked statement always throws an `OverflowException` if an integer calculation in the block overflows, as shown in this example:

```
int number = int.MaxValue;
checked
{
 int willThrow = number++;
 Console.WriteLine("This will not be reached");
}
```



**Important** Only integer arithmetic directly inside the checked block is subject to overflow checking. For example, if one of the checked statements is a method call, checking does not apply to code that runs in the called method.

You can also use the unchecked keyword to create an unchecked block statement. All integer arithmetic in an unchecked block is not checked and never throws an `OverflowException`. For example:

```
int number = int.MaxValue;
unchecked
{
 int wontThrow = number++;
 Console.WriteLine("This will be reached");
}
```

## Writing checked expressions

You can also use the checked and unchecked keywords to control overflow checking on integer expressions by preceding just the individual parenthesized expression with the checked or unchecked keyword, as shown in these examples:

```
int wontThrow = unchecked(int.MaxValue + 1);
int willThrow = checked(int.MaxValue + 1);
```

The compound operators (such as `+=` and `-=`) and the increment (`++`) and decrement (`--`) operators are arithmetic operators and can be controlled by using the checked and unchecked keywords. Remember, `x += y` is the same as `x = x + y`.



**Important** You cannot use the checked and unchecked keywords to control floating-point (noninteger) arithmetic. The checked and unchecked keywords apply only to integer arithmetic using data types such as `int` and `long`. Floating-point arithmetic never throws `OverflowException` exceptions—not even when you divide by 0.0. (Remember from Chapter 2 that .NET has a special floating-point representation for infinity.)

In the following exercise, you'll see how to perform checked arithmetic when using Visual Studio 2022.

## To use checked expressions

1. Return to Visual Studio 2022.

2. On the **Debug** menu, select **Start Debugging**.

You will now attempt to multiply two large values.

3. In the **Left Operand** box, type **9876543**. In the **Right Operand** box, type **9876543**.

4. Select the **\* Multiplication** button, and then select **Calculate**.

The value `-1195595903` appears in the Result box on the form. This is a negative value, which cannot possibly be correct. This value is the result of a multiplication operation that silently overflowed the 32-bit limit of the `int` type.

5. Return to Visual Studio and stop debugging.

6. In the Code and Text Editor window displaying `MainPage.xaml.cs`, locate the `multiplyValues` method, which should look like this:

```
private void multiplyValues()
{
 int lhs = int.Parse(lhsOperand.Text);
 int rhs = int.Parse(rhsOperand.Text);
 int outcome = 0;

 outcome = lhs * rhs;
 expression.Text = $"{lhs} * {rhs}";
 result.Text = outcome.ToString();
}
```

The statement `outcome = lhs * rhs;` contains the multiplication operation that's silently overflowing.

7. Edit this statement so that the calculation value is checked, like this:

```
outcome = checked(lhs * rhs);
```

The multiplication is now checked and will throw an `OverflowException` rather than silently returning the wrong answer.

8. On the **Debug** menu, select **Start Debugging**.

9. In the **Left Operand** box, type **9876543**. In the **Right Operand** box, type **9876543**.

10. Select the **\* Multiplication** button, and then select **Calculate**.

This time, Visual Studio drops into the debugger and reports that the multiplication resulted in an `OverflowException` exception. You must add a handler to catch this exception and handle it more gracefully than just failing with an error.

11. On the **Debug** menu, select **Stop Debugging**.

12. In the Code and Text Editor window displaying the MainPage.xaml.cs file, locate the `calculateClick` method.
13. Add the following catch handler (shown in bold) immediately after the existing `FormatException` catch handler in this method:

```
private void calculateClick(object sender, RoutedEventArgs e)
{
 try
 {
 ...
 }
 catch (FormatException fEx)
 {
 result.Text = fEx.Message;
 }
 catch (OverflowException oEx)
 {
 result.Text = oEx.Message;
 }
}
```

The logic of this catch handler is the same as that for the `FormatException` catch handler. However, it's still worth keeping these handlers separate instead of simply writing a generic `Exception` catch handler; in the future, you might decide to handle these exceptions differently.

14. On the **Debug** menu, select **Start Debugging** to build and run the application.
15. In the **Left Operand** box, type **9876543**. In the **Right Operand** box, type **9876543**.
16. Select the **\* Multiplication** button, and then select **Calculate**.

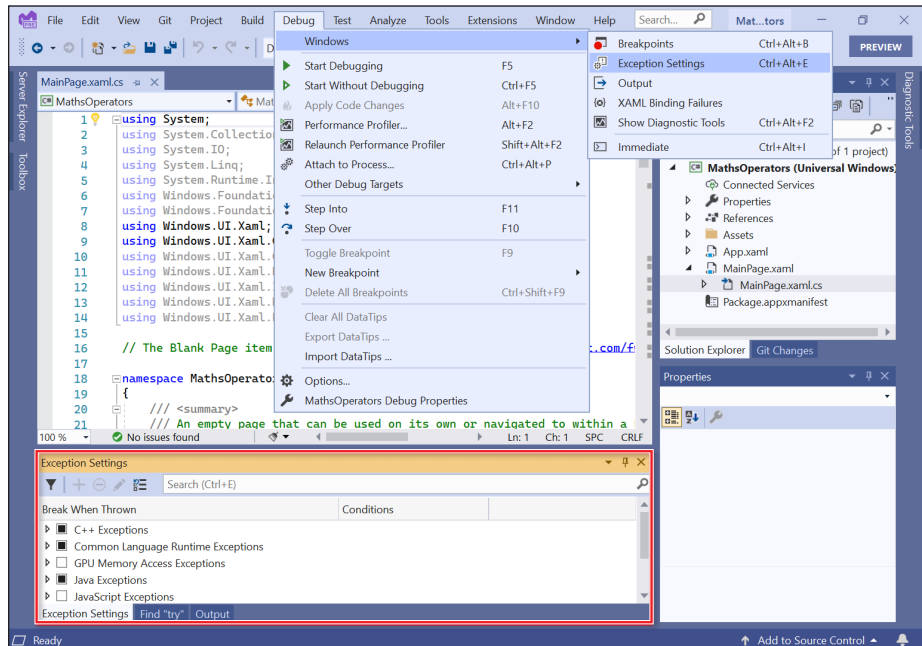
The second catch handler successfully catches the `OverflowException` and displays the message "Arithmetic operation resulted in an overflow" in the Result box.

17. Return to Visual Studio and stop debugging.

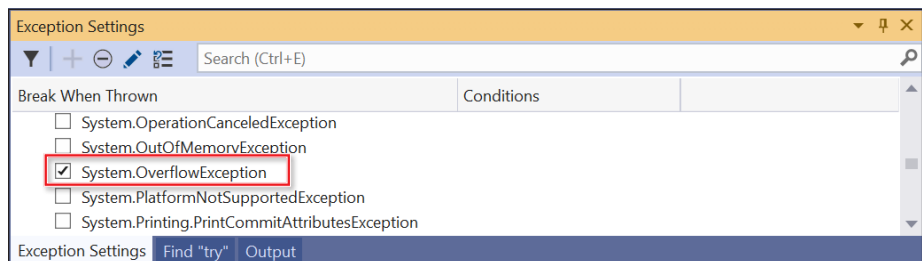
## Exception handling and the Visual Studio debugger

By default, the Visual Studio debugger only stops an application that's being debugged and reports exceptions that are unhandled. Sometimes, however, it's useful to be able to debug exception handlers themselves. In this case, you need to be able to trace exceptions when they are thrown by the application before they are caught. You can easily do this:

1. On the **Debug** menu, select **Windows**, and then select **Exception Settings**. The Exception Settings pane appears below the Code and Text Editor window.



2. In the Exception Settings pane, expand **Common Language Runtime Exceptions**, scroll down, and select **System.OverflowException**.



Now, when exceptions such as `OverflowException` occur, Visual Studio will drop into the debugger, and you can use the Step Into button on the Debug toolbar to step into the catch handler.

# Throwing exceptions

Suppose you're implementing a method called `monthName` that accepts a single `int` argument and returns the name of the corresponding month. For example, `monthName(1)` returns "January", `monthName(2)` returns "February", and so on. The question is, what should the method return if the integer argument is less than 1 or greater than 12? The best answer is that the method shouldn't return anything at all. Rather, it should throw an exception.

The .NET class libraries contain lots of exception classes specifically designed for situations such as this. Most often, you'll find that one of these classes describes your exception condition. If not, you can easily create your own exception class, but you need to know a bit more about the C# language before you can do that.

In this case, the existing .NET `ArgumentOutOfRangeException` class is just right. You can throw an exception by using a `throw` statement. The following example uses this technique. The `monthName` method uses a `switch` expression to convert a month number into a month name:

```
public string monthName(int month) => month switch {
 1 => "January",
 2 => "February",
 ...
 12 => "December",
 _ => throw new ArgumentOutOfRangeException("Bad month")
};
```

The `throw` statement needs an exception object to throw. This object contains the details of the exception, including any error messages. This example uses an expression that creates a new `ArgumentOutOfRangeException` object. The object is initialized with a string that populates its `Message` property by using a constructor. Constructors are covered in detail in Chapter 7, "Creating and managing classes and objects."

In the following exercises, you'll modify the `MathOperators` project to throw an exception if the user attempts to perform a calculation without selecting an option button for an operator.



**Note** This exercise is a little contrived, as any good application design would have a default option button selected initially, but this application is intended to illustrate a point.

## To throw an exception

1. Return to Visual Studio 2022.
2. On the **Debug** menu, select **Start Debugging**.
3. In the **Left Operand** box, type **24**. In the **Right Operand** box, type **36**. Then select **Calculate**.

Nothing appears in the Expression and Result boxes. The fact that you have not selected an operator option is not immediately obvious. It would be useful to write a diagnostic message in the Result box.

4. Return to Visual Studio and stop debugging.
5. In the Code and Text Editor window displaying MainPage.xaml.cs, locate and examine the `calculateClick` method, which should currently look like this:

```
private int calculateClick(object sender, RoutedEventArgs e)
{
 try
 {
 if ((bool)addition.IsChecked)
 {
 addValues();
 }
 else if ((bool)subtraction.IsChecked)
 {
 subtractValues();
 }
 else if ((bool)multiplication.IsChecked)
 {
 multiplyValues();
 }
 else if ((bool)division.IsChecked)
 {
 divideValues();
 }
 else if ((bool)remainder.IsChecked)
 {
 remainderValues();
 }
 }
 catch (FormatException fEx)
 {
 result.Text = fEx.Message;
 }
 catch (OverflowException oEx)
 {
 result.Text = oEx.Message;
 }
}
```

The addition, subtraction, multiplication, division, and remainder fields are the buttons that appear on the form. Each button has a property called `IsChecked` that indicates whether the user has selected it. The `IsChecked` property is a nullable Boolean that has the value `true` if the button is selected or `false` otherwise. (You'll learn more about nullable values in Chapter 8, "Understanding values and references.") The cascading `if` statement examines each button in turn to find which one is selected. (The option buttons are mutually exclusive, so the user can select only one option button.) If none of the buttons is selected, none of the `if` statements will be `true`, and none of the calculation methods will be called.

You could try to solve the problem by adding one more `else` statement to the `if-else` cascade to write a message to the Result box on the form, but a better solution is to separate the detection and signaling of an error from the catching and handling of that error.



6. Add another `else` statement to the end of the list of `if-else` statements and throw an `InvalidOperationException`, as shown in bold in the following code:

```
if ((bool)addition.IsChecked)
{
 addValues();
}
...
else if ((bool)remainder.IsChecked)
{
 remainderValues();
}
else
{
 throw new InvalidOperationException("No operator selected");
}
```

7. On the **Debug** menu, select **Start Debugging** to build and run the application.
8. In the **Left Operand** box, type **24**. In the **Right Operand** box, type **36**. Then select **Calculate**.

Visual Studio detects that your application has thrown an `InvalidOperationException`, and an Exception dialog opens. Your application has thrown an exception, but the code does not catch it yet.

9. On the **Debug** menu, select **Stop Debugging**.

Now that you have written a `throw` statement and verified that it throws an exception, you'll write a catch handler to handle this exception.

### To catch the exception

1. In the Code and Text Editor window displaying `MainPage.xaml.cs`, add the following catch handler shown in bold immediately below the two existing catch handlers in the `calculateClick` method:

```
...
catch (FormatException fEx)
{
 result.Text = fEx.Message;
}
catch (OverflowException oEx)
{
 result.Text = oEx.Message;
}
catch (InvalidOperationException ioEx)
{
 result.Text = ioEx.Message;
}
```

This code catches the `InvalidOperationException` that is thrown when the user fails to select an operator option button.

2. On the **Debug** menu, select **Start Debugging**.
3. In the **Left Operand** box, type **24**. In the **Right Operand** box, type **36**. Then select **Calculate**.

The message “No operator selected” appears in the Result box.



**Note** If your application drops into the Visual Studio debugger, you have probably enabled Visual Studio to catch all common language runtime (CLR) exceptions as they are thrown. If this happens, on the Debug menu, select Continue. Remember to disable Visual Studio from catching CLR exceptions as they are thrown when you have finished this exercise!

4. Return to Visual Studio and stop debugging.

The application is now a lot more robust. However, several exceptions could still arise that are not caught and will cause the application to fail. For example, if you attempt to divide by 0, an unhandled `DivideByZeroException` exception will be thrown. (Integer division by 0 does throw an exception, unlike floating-point division by 0.) One way to solve this problem is to write an ever-larger number of catch handlers inside the `calculateClick` method. Another solution is to add a general catch handler that catches `Exception` at the end of the list of catch handlers. This will trap all unexpected exceptions that you might have forgotten about, or that might be caused as a result of truly unusual circumstances.



**Note** Using a catchall handler to trap the `Exception` exception is not an excuse to omit catching specific exceptions. The more precise you can be in your exception handling, the easier it will be to maintain your code and spot the causes of any underlying or commonly recurring issues. Only use the `Exception` exception for cases that are really...well, exceptional.

In the following exercise, the “divide by zero” exception falls into this category. However, having established that this exception is a distinct possibility in a professional application, good practice would be to add a handler for the `DivideByZeroException` exception to the application.

## To catch unhandled exceptions

1. In the Code and Text Editor window displaying `MainPage.xaml.cs`, add the following catch handler to the end of the list of existing catch handlers in the `calculateClick` method:

```
catch (Exception ex)
{
 result.Text = ex.Message;
}
```

This catch handler will catch all as yet unhandled exceptions, whatever their specific type.



**Note** Be sure you add this handler to the end of the list. If you place it at the start, it will mask the other exception handlers.

2. On the **Debug** menu, select **Start Debugging**.

You will now attempt to perform some calculations known to cause exceptions and confirm that they are all handled correctly.

3. In the **Left Operand** box, type **24**. In the **Right Operand** box, type **36**.

4. Select **Calculate**.

Confirm that the diagnostic message "No operator selected" still appears in the Result box. This message was generated by the `InvalidOperationException` handler.

5. In the **Left Operand** box, type **John**, select the **+ Addition** button, and then select **Calculate**.

Confirm that the diagnostic message "Input string was not in a correct format" appears in the Result box. This message was generated by the `FormatException` handler.

6. In the **Left Operand** box, type **24**. In the **Right Operand** box, type **0**. Select the **/ Division** button, and then select **Calculate**.

Confirm that the diagnostic message "Attempted to divide by zero" appears in the Result box. This message was generated by the general `Exception` handler.

7. Experiment with other combinations of values and verify that exception conditions are handled without causing the application to stop.
8. When you've finished, return to Visual Studio and stop debugging.

## Using throw expressions

A throw exception is semantically similar to a throw statement, except it can be used anywhere you can use an expression. For example, suppose you want to set the `string` variable `name` to the value entered into the `nameField` box on a form, but only if the user has actually entered a value into that field; otherwise, you want to throw a "Missing input" exception. You could use the following code:

```
string name;
if (nameField.Text != "")
{
 name = nameField.Text;
}
else
{
 throw new Exception("Missing input"); // this is a throw statement
}
```

Although this code does the job, it's a little ungainly and verbose. You can simplify this block by using a `throw` expression together with another operator called the query-colon, or `?:`. The query-colon operator acts like an inline `if...else` statement for an expression. It is a ternary operator that takes the following three operands: a Boolean expression, an expression to evaluate and return if the Boolean expression is `true`, and another expression to evaluate and return if the Boolean expression is `false`. You can use it with a `throw` expression like this:

```
string name = nameField.Text !=
 "" ? nameField.Text : throw new Exception("Missing input"); // this is a throw expression
```

In this case, if the `nameField` box is not empty, the value of the `Text` property is stored in the `name` variable. Otherwise, the `throw` expression is evaluated, which in turn throws an exception. This code is much more concise than the previous example.

## Using a finally block

---

When an exception is thrown, it changes the flow of execution through the program. So, you can't guarantee that a statement will always run when the previous statement finishes because the previous statement might throw an exception. In this case, after the `catch` handler has run, the flow of control resumes at the next statement in the block holding this handler and not at the statement immediately following the code that raised the exception.

Look at the example that follows, which is adapted from the code in Chapter 5, "Using compound assignment and iteration statements." It's very easy to assume that the call to `reader.Dispose` will always occur when the `while` loop completes. After all, it's right there in the code.

```
TextReader reader = ...;
...
string line = reader.ReadLine();
while (line is not null)
{
 ...
 line = reader.ReadLine();
}
reader.Dispose();
```

Sometimes, it's not an issue if one particular statement does not run, but on many occasions it can be a big problem. If the statement releases a resource that was acquired in a previous statement, failing to execute this statement results in the resource being retained. This example is just such a case: when you open a file for reading, this operation acquires a resource (a file handle), and you must ensure that you call `reader.Dispose` to release the resource. If you don't, sooner or later you'll run out of file handles and be unable to open more files. (If you find that file handles are too trivial, think of database connections instead.)

The way to ensure that a statement is always run, whether or not an exception has been thrown, is to write that statement inside a `finally` block. A `finally` block occurs immediately after a `try` block or immediately after the last `catch` handler after a `try` block. As long as the program enters the `try` block associated with a `finally` block, the `finally` block will always run, even if an exception occurs. If an exception is thrown and caught locally, the exception handler executes first, followed by the `finally` block. If the exception is not caught locally (that is, the runtime has to search through the list of calling methods to find a handler), the `finally` block runs first. The important point is that the `finally` block always executes.

The solution to the `reader.Dispose` problem is as follows:

```
TextReader reader = ...;
...
try
{
 string line = reader.ReadLine();
 while (line is not null)
 {
 ...
 line = reader.ReadLine();
 }
}
finally
{
 if (reader is not null)
 {
 reader.Dispose();
 }
}
```

Even if an exception occurs while reading the file, the `finally` block ensures that the `reader.Dispose` statement always executes. You'll see another way to handle this situation in Chapter 14, "Using garbage collection and resource management."

## Summary

---

In this chapter, you learned how to catch and handle exceptions by using the `try` and `catch` constructs. You saw how to turn on and off integer overflow checking by using the `checked` and `unchecked` keywords. You learned how to throw an exception if your code detects an exception situation, and you saw how to use a `finally` block to ensure that critical code always runs, even if an exception occurs.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 7.
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

# Quick reference

To	Do this
Catch a specific exception	Write a catch handler that catches the specific exception class. For example: <pre>try {     ... } catch (FormatException fEx) {     ... }</pre>
Ensure that integer arithmetic is always checked for overflow	Use the checked keyword. For example: <pre>int number = Int32.MaxValue; checked { number++; }</pre>
Throw an exception	Use a throw statement. For example: <pre>throw new FormatException(source);</pre>
Catch all exceptions in a single catch handler	Write a catch handler that catches <code>Exception</code> . For example: <pre>try {     ... } catch (Exception ex) {     ... }</pre>
Ensure that some code will always run, even if an exception is thrown	Write the code within a <code>finally</code> block. For example: <pre>try {     ... } finally {     // always run }</pre>



## PART II

# Understanding the C# object model

<b>CHAPTER 7</b>	Creating and managing classes and objects . . . . .	181
<b>CHAPTER 8</b>	Understanding values and references . . . . .	205
<b>CHAPTER 9</b>	Creating value types with enumerations and structures . . . . .	231
<b>CHAPTER 10</b>	Using arrays . . . . .	251
<b>CHAPTER 11</b>	Understanding parameter arrays. . . . .	277
<b>CHAPTER 12</b>	Working with inheritance. . . . .	289
<b>CHAPTER 13</b>	Creating interfaces and defining abstract classes . . . . .	311
<b>CHAPTER 14</b>	Using garbage collection and resource management . . . . .	339



In Part I, you learned how to declare variables, use operators to create values, call methods, and write many of the statements you need when you implement a method. You now know enough to progress to the next stage: combining methods and data into your own functional data structures. The chapters in Part II show you how to do this.

In Part II, you'll learn about classes and structures, the two fundamental types that you use to model the entities and other items that constitute a typical C# application. In particular, you'll see how C# creates objects and value types based on the definitions of classes and structures, and how .NET manages the life cycle of these items. You'll find out how to create families of classes by using inheritance, and you'll learn how to aggregate items by using arrays.

# Creating and managing classes and objects

**After completing this chapter, you will be able to:**

- Understand the concepts of classification and encapsulation.
- Define a class containing a related set of methods and data items.
- Control the accessibility of members by using the `public` and `private` keywords.
- Write and call your own constructors.
- Create methods and data that can be shared by all instances of the same class by using the `static` keyword.
- Create anonymous classes.

Classes provide a convenient mechanism for modeling the entities manipulated by applications. An entity can represent a specific item, such as a customer, or something more abstract, such as a transaction. Part of the design process for any system focuses on determining the entities that are important to the processes that the system implements and then performing an analysis to see what information these entities need to hold and what operations they should perform. You store the information that a class holds as fields and use methods to implement the operations that a class can perform.



**Note** .NET uses classes extensively. The .NET libraries contain thousands of classes. You have used several of them already, including `Console` and `Exception`.

## Understanding classification

*Class* is the root word of the term *classification*. When you design a class, you systematically arrange information and behavior into a meaningful entity. This arranging is an act of classification, and it's something that everyone does—not just programmers. For example, all cars share common behaviors

(they can be steered, stopped, accelerated, and so on) and common attributes (they have a steering wheel, an engine, and so on). People use the word *car* to mean an object that shares these common behaviors and attributes. As long as everyone agrees on what a word means, this system works well, and you can express complex but precise ideas in a concise form. Without classification, it's hard to imagine how people could think or communicate at all.

Given that classification is so deeply ingrained in the way we think and communicate, it makes sense to try to write programs by classifying the different concepts inherent in a problem and its solution and then modeling these classes in a programming language. This is exactly what you can do with object-oriented programming languages, including Microsoft Visual C#.

## The purpose of encapsulation

---

Encapsulation is an important principle when defining classes. The idea is that a program that uses a class should not have to account for how that class actually works internally; the program simply creates an instance of a class and calls the methods of that class. As long as those methods do what they are designed to do, the program doesn't need to know how they are implemented. For example, when you call the `Console.WriteLine` method, you don't want to be bothered with all the intricate details of how the `Console` class physically arranges for data to be written to the screen. A class might need to maintain all sorts of internal state information to perform its various methods. This additional state information and activity is hidden from the program that's using the class. Therefore, encapsulation is sometimes referred to as information hiding. Encapsulation actually has two purposes:

- To combine methods and data within a class—in other words, to support classification
- To control the accessibility of the methods and data—in other words, to control the use of the class

## Defining and using a class

---

In C#, you use the `class` keyword to define a new class. The data and methods of the class occur in the body of the class between a pair of braces. Following is a C# class called `Circle` that contains one method (to calculate the circle's area) and one piece of data (the circle's radius):

```
class Circle
{
 int radius;
 double Area()
 {
 return Math.PI * radius * radius;
 }
}
```



**Note** The `Math` class contains methods for performing mathematical calculations and fields containing mathematical constants. The `Math.PI` field contains the value 3.14159265358979, which approximates the value of pi.

The body of a class contains ordinary methods (such as `Area`) and fields (such as `radius`). Recall that variables in a class are called fields. Chapter 2, “Working with variables, operators, and expressions,” showed how to declare variables, and Chapter 3, “Writing methods and applying scope,” demonstrated how to write methods, so there’s almost no new syntax here.

You can use the `Circle` class like you have used the other types you have already met. You create a variable specifying `Circle` as its type, and then you initialize the variable with some valid data. Here’s an example:

```
Circle c; // Create a Circle variable
c = new Circle(); // Initialize it
```

A point worth highlighting in this code is the use of the `new` keyword. Previously, when you initialized a variable such as an `int` or a `float`, you simply assigned it a value:

```
int i;
i = 42;
```

You cannot do the same with variables of class types. One reason for this is that C# just doesn’t provide the syntax for assigning literal class values to variables. You cannot write a statement such as this:

```
Circle c;
c = 42;
```

After all, what is the `Circle` equivalent of 42? Another reason concerns the way in which memory for variables of class types is allocated and managed by the runtime. This is discussed further in Chapter 8, “Understanding values and references.” For now, just accept that the `new` keyword creates a new instance of a class, more commonly called an object.

You can, however, directly assign an instance of a class to another variable of the same type, like this:

```
Circle c;
c = new Circle();
Circle d;
d = c;
```

However, this is not as straightforward as it might first appear, for reasons that are described in Chapter 8.



**Important** Don’t confuse the terms *class* and *object*. A class is the definition of a type. An object is an instance of that type created when the program runs. Several different objects can be instances of the same class.

## Type inference and the new operator

In the examples shown in the main body of the text, the class name, `Circle`, in the new expression is actually superfluous. You can simply write:

```
Circle c;
c = new();
```

The C# compiler understands that `c` is a `Circle` variable, so it will create a new `Circle` object by default. Whether you include or omit the type in the new expression is often a matter of personal preference, but there will be times when you need to specify the type in the new expression. (You'll meet some examples later.)

On the other hand, if you combine the declaration and initialization into a single statement, omitting the type can make your code more concise. For example:

```
Circle c = new();
```

Alternatively, you can declare the `c` variable as a `var` and specify the type for the new operator (this is my preferred style):

```
var c = new Circle();
```

## Controlling accessibility

Surprisingly, the `Circle` class is currently of no practical use. By default, when you encapsulate your methods and data within a class, the class forms a boundary to the outside world. Fields (such as `radius`) and methods (such as `Area`) defined in the class can be used by other methods inside the class but not by the outside world; they are private to the class. So, although you can create a `Circle` object in a program, you cannot access its `radius` field or call its `Area` method, which is why the class is not yet of much use! However, you can modify the definition of a field or method with the `public` and `private` keywords to control whether it is accessible from the outside:

- A method or field is private if it is accessible only from within the class. To declare that a method or field is private, you write the keyword `private` before its declaration. As intimated previously, this is actually the default, but it's good practice to state explicitly that fields and methods are private to avoid any confusion.
- A method or field is public if it is accessible both within and from outside the class. To declare that a method or field is public, you add the keyword `public` before its declaration.

Here is the `Circle` class again. This time, `Area` is declared as a public method, and `radius` is declared as a private field:

```
class Circle
{
 private int radius;
 public double Area()
}
```

```

{
 return Math.PI * radius * radius;
}
}

```



**Note** If you are a C++ programmer, be aware that no colon appears after the `public` and `private` keywords. You must repeat the keyword for every field and method declaration.

Although `radius` is declared as a private field and is not accessible from outside the class, `radius` is accessible within the `Circle` class. The `Area` method is inside the `Circle` class, so the body of `Area` has access to `radius`. However, the class is still of limited value because there is no way to initialize the `radius` field. To fix this, you can use a constructor.



**Tip** Variables declared in a method are not initialized by default. However, the fields in a class are automatically initialized to 0, `false`, or `null`, depending on their type. Nonetheless, it's still good practice to provide an explicit means of initializing fields.

## Naming and accessibility

Many organizations have their own house style that they ask developers to follow when they write code. Part of this style often involves rules for naming identifiers. Typically, the purpose of these rules is to make the code easier to maintain. The following recommendations are reasonably common and relate to the naming conventions for fields and methods based on the accessibility of class members. However, C# does not enforce these rules:

- Identifiers that are public should start with a capital letter. For example, `Area` starts with `A` (not `a`) because it's public. This system is known as the `PascalCase` naming scheme (because it was first used in the Pascal language).
- Identifiers that are not public (which include local variables) should start with a lowercase letter. For example, `radius` starts with `r` (not `R`) because it's private. This system is known as the `camelCase` naming scheme.



**Note** Some organizations use the `camelCase` scheme only for methods and adopt the convention to name private fields starting with an underscore character, such as `_radius`. However, the examples in this book use `camelCase` naming for private methods and fields.

There's only one exception to this rule: class names should start with a capital letter, and constructors must match the name of their class exactly; therefore, a private constructor must start with a capital letter.



**Important** Don't declare two public class members whose names differ only in case. If you do, developers using other languages that are not case sensitive (such as Microsoft Visual Basic) might not be able to integrate your class into their solutions.

## Working with constructors

When you use the `new` keyword to create an object, the runtime needs to construct that object by using the definition of the class. The runtime must grab a piece of memory from the operating system, fill it with the fields defined by the class, and then invoke a constructor to perform any initialization required.

A constructor is a special method that runs automatically when you create an instance of a class. It has the same name as the class, and it can take parameters, but it cannot return a value (not even `void`). Every class must have a constructor. If you don't write one, the compiler automatically generates a default constructor for you (although the compiler-generated default constructor doesn't actually do anything).

You can write your own default constructor quite easily. Just add a `public` method that does not return a value and give it the same name as the class. The following example shows the `Circle` class with a default constructor that initializes the `radius` field to 0:

```
class Circle
{
 private int radius;
 public Circle() // default constructor
 {
 radius = 0;
 }

 public double Area()
 {
 return Math.PI * radius * radius;
 }
}
```



**Note** In C# parlance, the default constructor is a constructor that doesn't take any parameters. Regardless of whether the compiler generates the default constructor or you write it yourself, a constructor that does not take any parameters is still the default constructor. You can also write nondefault constructors (constructors that do take parameters), as you'll see in the next section, "Overloading constructors."

In this example, the constructor is marked `public`. If this keyword is omitted, the constructor will be `private` (just like any other method and field). If the constructor is `private`, it cannot be used outside the class, which prevents you from being able to create `Circle` objects from methods that are not part of the `Circle` class. You might, therefore, think that `private` constructors are not that valuable. They do have their uses, but they are beyond the scope of the current discussion.

Having added a `public` constructor, you can now use the `Circle` class and exercise its `Area` method. Notice how you use dot notation to invoke the `Area` method on a `Circle` object:

```
Circle c;
c = new Circle();
double areaOfCircle = c.Area();
```

## Overloading constructors

You can now declare a `Circle` variable, use it to reference a newly created `Circle` object, and then call its `Area` method. However, there is one last problem: The area of all `Circle` objects will always be 0, because the default constructor sets the `radius` to 0. Moreover, the `radius` field is private, and there is no easy way to change its value after it has been initialized.

A constructor is just a special kind of method, and like all methods, it can be overloaded. Just as there are several versions of the `Console.WriteLine` method, each of which takes different parameters, you can write different versions of a constructor. So, you can add another constructor to the `Circle` class with a parameter that specifies what radius to use, like this:

```
class Circle
{
 private int radius;
 public Circle() // default constructor
 {
 radius = 0;
 }

 public Circle(int initialRadius) // overloaded constructor
 {
 radius = initialRadius;
 }

 public double Area()
 {
 return Math.PI * radius * radius;
 }
}
```



**Note** The order of the constructors in a class is immaterial; you can define constructors in the order you feel most comfortable with.

You can then use this constructor when you create a new `Circle` object, such as in the following:

```
var c = new Circle(45);
```

When you build the application, the compiler determines which constructor it should call based on the parameters that you specify to the `new` operator. In this example, you passed an `int`, so the compiler generates code that invokes the constructor that takes an `int` parameter.



You should be aware of an important feature of the C# language: if you write your own constructor for a class, the compiler does not generate a default constructor. Therefore, if you've written your own constructor that accepts one or more parameters and you also want a default constructor, you'll have to write the default constructor yourself.

## Partial classes

A class can contain a number of methods, fields, and constructors, as well as other items discussed in later chapters. So, a highly functional class can become quite large. With C#, you can split the source code for a class into separate files so you can organize the definition of a large class into smaller pieces that are easier to manage. This feature is used by Visual Studio 2022 for Universal Windows Platform (UWP) apps, which maintains the source code that the developer can edit in a separate file from the code that's generated by Visual Studio whenever the layout of a form changes.

When you split a class across multiple files, you define the parts of the class by using the `partial` keyword in each file. For example, if the `Circle` class is split between two files called `circ1.cs` (containing the constructors) and `circ2.cs` (containing the methods and fields), the contents of `circ1.cs` look like this:

```
partial class Circle
{
 public Circle() // default constructor
 {
 this.radius = 0;
 }

 public Circle(int initialRadius) // overloaded constructor
 {
 this.radius = initialRadius; }
}
```

As for the contents of `circ2.cs`, they look like this:

```
partial class Circle
{
 private int radius;
 public double Area()
 {
 return Math.PI * this.radius * this.radius;
 }
}
```

When you compile a class that's been split into separate files, you must provide all the files to the compiler.

In the following exercise, you'll declare a class that models a point in two-dimensional space. The class will contain two `private` fields for holding the x- and y-coordinates of a point and will provide constructors for initializing these fields. You'll create instances of the class by using the `new` keyword and calling the constructors.

### To write constructors and create objects

1. Start Visual Studio 2022, if it is not already running.
2. Open the **Classes** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 7\Classes** folder in your **Documents** folder.
3. In Solution Explorer, double-click the file `Program.cs` to display it in the Code and Text Editor window.
4. In the `Program` class, locate the `Main` method.

The `Main` method calls the `doWork` method, which is wrapped in a `try` block and followed by a `catch` handler. With this `try/catch` block, you can write the code that would typically go inside `Main` in the `doWork` method instead, and be safe in the knowledge that it will catch and handle any exceptions. The `doWork` method currently contains nothing but a `// TODO: comment`.

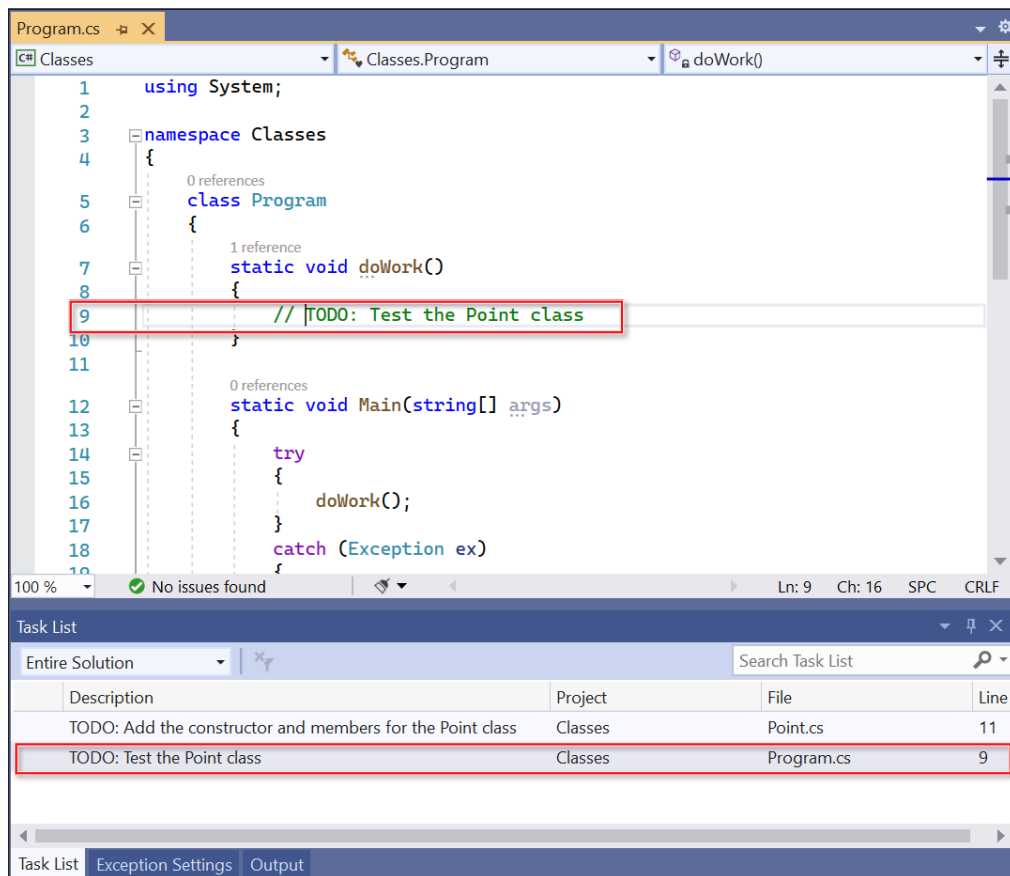
```
class Program
{
 static void doWork()
 {
 // TODO: Test the Point class:
 }

 static void Main(string[] args)
 {
 try
 {
 doWork();
 }
 catch (Exception ex)
 {
 Console.WriteLine(ex.Message);
 }
 }
}
```

## // TODO: comments

Developers frequently add `// TODO:`  comments as a reminder to revisit a piece of code. These comments often contain a description of the work to be performed, such as `// TODO: Implement the doWork method.`

Visual Studio recognizes this form of comment, and you can quickly locate them anywhere in an application by using the Task List window. To display this window, on the View menu, select Task List. The Task List window opens below the Code and Text Editor window by default. All the TODO comments will be listed. You can then double-click any of these comments to go directly to the corresponding code, which will be displayed in the Code and Text Editor window.



5. Display the file `Point.cs` in the Code and Text Editor window.

This file defines a class called `Point`, which you'll use to represent the location of a point in two-dimensional space, defined by a pair of `x`- and `y`-coordinates. The `Point` class is currently empty apart from another `// TODO: comment`.

6. Return to the `Program.cs` file. In the `Program` class, edit the body of the `doWork` method, and replace the `// TODO: comment` with the following statement:

```
var origin = new Point();
```

This statement creates a new instance of the `Point` class and invokes its default constructor.

7. On the **Build** menu, select **Build Solution**.

The code builds without error because the compiler automatically generates the code for a default constructor for the `Point` class. However, you cannot see the C# code for this constructor because the compiler does not generate any source language statements.

8. Return to the `Point` class in the file `Point.cs`. Replace the `// TODO: comment` with a `public` constructor that accepts two `int` arguments, called `x` and `y`, and that calls the `Console.WriteLine` method to display the values of these arguments to the console, as shown in bold type in the following code example:

```
class Point
{
 public Point(int x, int y)
 {
 Console.WriteLine($"x:{x}, y:{y}");
 }
}
```

9. On the **Build** menu, select **Build Solution**.

The compiler now reports an error:

```
There is no argument that corresponds to the required formal parameter 'x' of
'Point.Point(int, int)'
```

What this rather verbose message means is that the call to the default constructor in the `doWork` method is now invalid because there is no longer a default constructor. You have written your own constructor for the `Point` class, so the compiler doesn't generate the default constructor. You'll now fix this by writing your own default constructor.

10. Edit the `Point` class by adding a `public` default constructor that calls `Console.WriteLine` to write the string "Default constructor called" to the console, as shown in bold type in the example that follows. The `Point` class should now look like this:

```
class Point
{
 public Point()
 {
 Console.WriteLine("Default constructor called");
 }

 public Point(int x, int y)
 {
 Console.WriteLine($"x:{x}, y:{y}");
 }
}
```

11. On the **Build** menu, select **Build Solution**.

The program should now build successfully.

12. In the `Program.cs` file, edit the body of the `doWork` method. Declare a variable called `bottomRight` of type `Point`, and initialize it to a new `Point` object by using the constructor with two arguments, as shown in bold type in the code that follows. Supply the values 2496 and 1664, representing the coordinates at the lower-right corner of the screen based on the resolution 2496 × 1664 (the resolution for a Microsoft Surface 3 Laptop with a 15-inch screen). The `doWork` method should now look like this:

```
static void doWork()
{
 var origin = new Point();
 var bottomRight = new Point(2496, 1664);
}
```

13. On the **Debug** menu, select **Start Without Debugging**.

The program builds and runs, displaying the following messages to the console:

```
Default constructor called
x:2496, y:1664
```

14. Press the **Enter** key to end the program and return to Visual Studio 2022.

You'll now add two `int` fields to the `Point` class to represent the x- and y-coordinates of a point, and you'll modify the constructors to initialize these fields.

15. Edit the `Point` class in the `Point.cs` file and add two private fields, called `x` and `y`, of type `int`, as shown in bold type in the code that follows. The `Point` class should now look like this:

```
class Point
{
 private int x, y;
```

```

 public Point()
 {
 Console.WriteLine("default constructor called");
 }

 public Point(int x, int y)
 {
 Console.WriteLine($"x:{x}, y:{y}");
 }
}

```

You'll modify the second `Point` constructor to initialize the `x` and `y` fields to the values of the `x` and `y` parameters. However, there's a potential trap when you do this: if you're not careful, the constructor could look like this:

```

public Point(int x, int y) // Don't type this!
{
 x = x;
 y = y;
}

```

Although this code will compile, these statements appear to be ambiguous. How does the compiler know in the statement `x = x;` that the first `x` is the field and the second `x` is the parameter? The answer is that it doesn't! A method parameter with the same name as a field hides the field for all statements in the method. All this code actually does is assign the parameters to themselves; it does not modify the fields at all. This is clearly not what you want.

The solution is to use the `this` keyword to qualify which variables are parameters and which are fields. Prefixing a variable with `this` means "the field in this object."

16. Modify the `Point` constructor that takes two parameters by replacing the `Console.WriteLine` statement with the following code shown in bold type:

```

public Point(int x, int y)
{
 this.x = x;
 this.y = y;
}

```

17. Edit the default `Point` constructor to initialize the `x` and `y` fields to `-1`, as follows in bold type. Note that although there are no parameters to cause confusion (and Visual Studio will pop up a tooltip stating that you don't need to use `this`), it's still good practice to qualify the field references in this way:

```

public Point()
{
 this.x = -1;
 this.y = -1;
}

```

18. On the **Build** menu, select **Build Solution**. Confirm that the code compiles without errors or warnings. (You can run it, but it does not produce any output yet.)

Methods that belong to a class and that operate on the data belonging to a particular instance of a class are called instance methods. (You'll learn about other types of methods later in this chapter.) In the following exercise, you'll write an instance method for the `Point` class, called `DistanceTo`, which calculates the distance between two points.

### To write and call instance methods

1. In the `Classes` project in Visual Studio 2022, add the following `public` instance method called `DistanceTo` to the `Point` class after the constructors. The method accepts a single `Point` argument called `other` and returns a `double`.

The `DistanceTo` method should look like this:

```
class Point
{
 ...
 public double DistanceTo(Point other)
 {
 }
}
```

In the following steps, you'll add code to the body of the `DistanceTo` instance method to calculate and return the distance between the `Point` object being used to make the call and the `Point` object passed as a parameter. To do this, you must calculate the difference between the x-coordinates and the y-coordinates.

2. In the `DistanceTo` method, declare a local `int` variable called `xDiff` and initialize it with the difference between `this.x` and `other.x`, as shown here in bold type:

```
public double DistanceTo(Point other)
{
 int xDiff = this.x - other.x;
}
```

3. Declare another local `int` variable called `yDiff` and initialize it with the difference between `this.y` and `other.y`, as shown here in bold type:

```
public double DistanceTo(Point other)
{
 int xDiff = this.x - other.x;
 int yDiff = this.y - other.y;
}
```



**Note** Although the `x` and `y` fields are private, other instances of the same class can still access them. It's important to understand that the term *private* operates at the class level and not at the object level; two objects that are instances of the same class can access each other's private data, but objects that are instances of another class cannot.

To calculate the distance, you can use Pythagoras' theorem and compute the square root of the sum of the square of `xDiff` and the square of `yDiff`. The `System.Math` class provides the `Sqrt` method, which you can use to calculate square roots.

4. Declare a variable called `distance` of type `double` and use it to hold the result of the calculation just described.

```
public double DistanceTo(Point other)
{
 int xDiff = this.x - other.x;
 int yDiff = this.y - other.y;
 double distance = Math.Sqrt((xDiff * xDiff) + (yDiff * yDiff));
}
```

5. Add a return statement to the end of the `DistanceTo` method and return the value in the `distance` variable:

```
public double DistanceTo(Point other)
{
 int xDiff = this.x - other.x;
 int yDiff = this.y - other.y;
 double distance = Math.Sqrt((xDiff * xDiff) + (yDiff * yDiff));
 return distance;
}
```

You'll now test the `DistanceTo` method.

6. Return to the `doWork` method in the `Program` class. After the statements that declare and initialize the `origin` and `bottomRight` `Point` variables, declare a variable called `distance` of type `double`.
7. Initialize the `double` variable with the result obtained when you call the `DistanceTo` method on the `origin` object, passing the `bottomRight` object to it as an argument.

The `doWork` method should now look like this:

```
static void doWork()
{
 Point origin = new Point();
 Point bottomRight = new Point(2496, 1664);
 double distance = origin.DistanceTo(bottomRight);
}
```



**Note** Microsoft IntelliSense should display the `DistanceTo` method when you type the period character after `origin`.



8. Add to the `doWork` method another statement that writes the value of the `distance` variable to the console by using the `Console.WriteLine` method.

The completed `doWork` method should look like this:

```
static void doWork()
{
 Point origin = new Point();
 Point bottomRight = new Point(2496, 1664);
 double distance = origin.DistanceTo(bottomRight);
 Console.WriteLine($"Distance is: {distance}");
}
```

9. On the **Debug** menu, select **Start Without Debugging**.
10. Confirm that the value `1568.4546534726467` is written to the console window and then press **Enter** to close the application and return to Visual Studio 2022.

## Deconstructing an object

You use a constructor to create and initialize an object, typically by populating any fields it contains. A deconstructor enables you to examine an object and extract the values of its fields. Taking the `Point` class from the previous exercise as an example, you can implement a deconstructor that retrieves the values of the `x` and `y` fields, like this:

```
class Point
{
 private int x, y;
 ...

 public void Deconstruct(out int x, out int y)
 {
 x = this.x;
 y = this.y;
 }
}
```

You should note the following points about a deconstructor:

- It is always named `Deconstruct`.
- It must be a `void` method.
- It must take one or more parameters. These parameters will be populated with the values from the fields in the objects.
- The parameters are marked with the `out` modifier. This means that if you assign values to them, these values will be passed back to the caller. (You'll learn more about `out` modifiers in Chapter 8.)
- The code in the body of the method assigns the values to be returned to the parameters.

You call the deconstructor in a manner similar to that used to call a method that returns a tuple (described in Chapter 3). You simply create a tuple and assign an object to it, like this:

```
Point origin = new Point();
...
(int xVal, int yVal) = origin;
```

Behind the scenes, C# runs the deconstructor and passes it the variables defined in the tuple as the parameters. The code in the deconstructor populates these variables. Assuming you have not modified the default constructor for the `Point` class, the `xVal` and `yVal` variables should both contain the value `-1`.



**Note** In addition to deconstructors, there are other ways to retrieve the values held by the fields in an object. Chapter 15, “Implementing properties to access fields,” describes another very common strategy.

## Understanding static methods and data

In the preceding exercise, you used the `Sqrt` method of the `Math` class. Similarly, when looking at the `Circle` class, you read the `PI` field of the `Math` class. If you think about it, the way in which you called the `Sqrt` method or read the `PI` field was slightly odd. You invoked the method or read the field on the class itself, not on an object of type `Math`. It’s like trying to write `Point.DistanceTo` rather than `origin.DistanceTo` in the code you added in the preceding exercise. So, what’s happening, and how does this work?

Not all methods naturally belong to an instance of a class. They are utility methods; they provide a useful function that’s independent of any specific class instance. The `WriteLine` method of the `Console` class that’s been used extensively throughout this book is one common example. The `Sqrt` method is another. If `Sqrt` were an instance method of `Math`, you’d have to create a `Math` object on which to call `Sqrt`:

```
Math m = new Math();
double d = m.Sqrt(42.24);
```

This would be cumbersome. The `Math` object would play no part in the calculation of the square root. All the input data that `Sqrt` needs is provided in the parameter list, and the result is passed back to the caller by using the method’s return value. Objects are not really needed here, so forcing `Sqrt` into an instance straitjacket is just not a good idea.



**Note** In addition to containing the `Sqrt` method and the `PI` field, the `Math` class contains many other mathematical utility methods, such as `Sin`, `Cos`, `Tan`, and `Log`.

In C#, all methods must be declared within a class. However, if you declare a method or a field as static, you can call the method or access the field by using the name of the class. No instance is required. This is how the `Sqrt` method of the `Math` class is declared:

```
class Math
{
 public static double Sqrt(double d)
 {
 ...
 }
 ...
}
```

You can invoke the `Sqrt` method like this:

```
double d = Math.Sqrt(42.24);
```

A static method does not depend on an instance of the class, and it cannot access any instance fields or instance methods defined in the class; it can use only fields and other methods that are marked as static.

## Creating a shared field

Defining a field as static makes it possible to create a single instance of a field that's shared among all objects created from a single class. (Nonstatic fields are local to each instance of an object.) In the following example, the static field `NumCircles` in the `Circle` class is incremented by the `Circle` constructor every time a new `Circle` object is created:

```
class Circle
{
 private int radius;
 public static int NumCircles = 0;

 public Circle() // default constructor
 {
 radius = 0;
 NumCircles++;
 }

 public Circle(int initialRadius) // overloaded constructor
 {
 radius = initialRadius;
 NumCircles++;
 }
}
```

All `Circle` objects share the same instance of the `NumCircles` field, so the statement `NumCircles++` increments the same data every time a new instance is created. Notice that you cannot prefix `NumCircles` with the `this` keyword because `NumCircles` does not belong to a specific object.

You can access the `NumCircles` field from outside the class by specifying the `Circle` class rather than a `Circle` object, such as in the following example:

```
Console.WriteLine($"Number of Circle objects: {Circle.NumCircles}");
```



**Note** Static methods are also called class methods. However, static fields aren't usually called class fields; they're just called static fields (or sometimes static variables).

## Creating a static field by using the const keyword

By prefixing the field with the `const` keyword, you can declare that a field is static but that its value can never change. The keyword `const` is short for constant. A `const` field doesn't use the `static` keyword in its declaration but is nevertheless static. However, for reasons that are beyond the scope of this book, you can declare a field as `const` only when the field is a numeric type (such as `int` or `double`), a string, or an enumeration. (You'll learn about enumerations in Chapter 9, "Creating value types with enumerations and structures.") For example, here's how the `Math` class declares `PI` as a `const` field:

```
class Math
{
 ...
 public const double PI = 3.14159265358979;
}
```

## Understanding static classes

Another feature of the C# language is the ability to declare a class as static. A static class can contain only static members. (All objects that you create by using the class share a single copy of these members.)

The purpose of a static class is purely to act as a holder of utility methods and fields. A static class cannot contain any instance data or methods, and it doesn't make sense to try to create an object from a static class by using the `new` operator. In fact, you can't actually use `new` to create an instance of an object using a static class even if you want to. (The compiler will report an error if you try.) If you need to perform any initialization, a static class can have a default constructor as long as it is also declared as static. Any other types of constructor are illegal and will be reported as such by the compiler.

If you were defining your own version of the `Math` class, one containing only static members, it could look like this:

```
public static class Math
{
 public static double Sin(double x) {...}
 public static double Cos(double x) {...}
 public static double Sqrt(double x) {...}
 ...
}
```



**Note** The real `Math` class is not defined this way because it actually does have some instance methods.

## Static using statements

Whenever you call a static method or reference a static field, you must specify the class to which the method or field belongs, such as `Math.Sqrt` or `Console.WriteLine`. Static `using` statements enable you to bring a class into scope and omit the class name when accessing static members. They operate in much the same way as ordinary `using` statements that bring namespaces into scope. The following example illustrates how to use them:

```
using static System.Math;
using static System.Console;
...
var root = Sqrt(99.9);
WriteLine($"The square root of 99.9 is {root}");
```

Note the use of the keyword `static` in the `using` statements. The example brings the static methods of the `System.Math` and `System.Console` classes into scope. (You must fully qualify the classes with their namespaces.) You can then simply call the `Sqrt` and `WriteLine` methods. The compiler determines which class each method belongs to.

Herein lies a potential maintenance issue. Although you're typing less code, you have to balance this with the additional effort required when someone else has to maintain your code, because it's no longer clear which class each method belongs to. IntelliSense in Visual Studio helps to some extent, but to a developer reading through the code and trying to track down the cause of a bug, it can obfuscate matters. So, use static `using` statements carefully. (My preferred style is not to use them, although you are free to make your own choice!)

In the final exercise in this chapter, you'll add a private `static` field to the `Point` class and initialize the field to 0. You will increment this count in both constructors. Finally, you'll write a public `static` method to return the value of this private `static` field. With this field, you can discover how many `Point` objects you have created.

### To write static members and call static methods

1. In Visual Studio 2022, display the `Point` class in the Code and Text Editor window.
2. Add a private `static` field called `objectCount` of type `int` to the `Point` class immediately before the first constructor and initialize it, like this:

```
class Point
{
 private int x, y;
 private static int objectCount = 0;

 public Point()
 {
 ...
 }
 ...
}
```



**Note** You can write the keywords `private` and `static` in any order when you declare a field such as `objectCount`. However, the preferred order is `private` first, `static` second.

3. Add a statement to both `Point` constructors to increment the `objectCount` field, as shown in bold type in the code example that follows.

The `Point` class should now look like this:

```
class Point
{
 private int x, y;
 private static int objectCount = 0;

 public Point()
 {
 this.x = -1;
 this.y = -1;
 objectCount++;
 }

 public Point(int x, int y)
 {
 this.x = x;
 this.y = y;
 objectCount++;
 }
 ...
}
```

Each time an object is created, its constructor is called. As long as you increment the `objectCount` in each constructor (including the default constructor), `objectCount` will hold the number of objects created so far. This strategy works only because `objectCount` is a shared static field. If `objectCount` were an instance field, each object would have its own personal `objectCount` field that would be set to 1.

The question now is this: How can users of the `Point` class find out how many `Point` objects have been created? At the moment, the `objectCount` field is private and not available outside the class. A poor solution would be to make the `objectCount` field public. This strategy would break the encapsulation of the class, and you would then have no guarantee that the `objectCount` field's value was correct because anyone could change the value in the field. A much better idea is to provide a public static method that returns the value of the `objectCount` field. This is what you'll do now.

4. Add a public static method to the end of `Point` class called `ObjectCount` that returns an `int` but does not take any parameters. This method should return the value of the `objectCount` field, as shown in bold type here:

```
class Point
{
 ...
 public static int ObjectCount() => objectCount;
}
```

5. Display the Program class in the Code and Text Editor window. Add a statement to the `doWork` method to write the value returned from the `ObjectCount` method of the `Point` class to the screen, as shown in bold type in the following code example:

```
static void doWork()
{
 Point origin = new Point();
 Point bottomRight = new Point(2496, 1664);
 double distance = origin.distanceTo(bottomRight);
 Console.WriteLine($"Distance is: {distance}");
 Console.WriteLine($"Number of Point objects: {Point.ObjectCount()}");
}
```

The `ObjectCount` method is called by referencing `Point`, the name of the class, and not the name of a `Point` variable (such as `origin` or `bottomRight`). Because two `Point` objects have been created by the time `ObjectCount` is called, the method should return the value `2`.

6. On the **Debug** menu, select **Start Without Debugging**.

Confirm that the message "Number of Point objects: 2 " is written to the console window (after the message displaying the value of the distance variable).

7. Press **Enter** to close the program and return to Visual Studio 2022.

## Anonymous classes

---

An anonymous class has no name. This sounds rather strange, but it's actually quite handy in some situations that you'll see later in this book, especially when using query expressions. (You'll learn about query expressions in Chapter 20, "Decoupling application logic and handling events.") For now, you'll have to take it on faith that they are useful.

You create an anonymous class simply by using the `new` keyword and a pair of braces defining the fields and values that you want the class to contain, like this:

```
myAnonymousObject = new { Name = "John ", Age = 47 };
```

This class contains public fields called `Name` (initialized to the string "John") and `Age` (initialized to the integer 47). The compiler infers the types of the fields from the types of the data you specify to initialize them.

When you define an anonymous class, the compiler synthesizes its own name for the class, but it won't tell you what it is. Anonymous classes, therefore, raise a potentially interesting conundrum: If you don't know the name of the class, how can you create an object of the appropriate type and assign an instance of the class to it? In the code example shown earlier, what should the type of the variable `myAnonymousObject` be? The answer is that you don't know, and that is the point of anonymous classes! However, this isn't a problem if you declare `myAnonymousObject` as an implicitly typed variable by using the `var` keyword, like this:

```
var myAnonymousObject = new { Name = "John ", Age = 47 };
```

Remember that the `var` keyword causes the compiler to create a variable of the same type as the expression used to initialize it. In this case, the type of the expression is whatever name the compiler happens to generate for the anonymous class.

You can access the fields in the object by using the familiar dot notation, as demonstrated here:

```
Console.WriteLine($"Name: {myAnonymousObject.Name} Age: {myAnonymousObject.Age}");
```

You can even create other instances of the same anonymous class but with different values, such as in the following:

```
var anotherAnonymousObject = new { Name = "Diana ", Age = 53 };
```

The C# compiler uses the names, types, number, and order of the fields to determine whether two instances of an anonymous class have the same type. In this case, the variables `myAnonymousObject` and `anotherAnonymousObject` have the same number of fields, with the same name and type, in the same order, so both variables are instances of the same anonymous class. This means that you can perform assignment statements such as this:

```
anotherAnonymousObject = myAnonymousObject;
```



**Note** Be warned that this assignment statement might not accomplish what you expect! You'll learn more about assigning object variables in Chapter 8.

There are quite a few restrictions on the contents of an anonymous class. For example, anonymous classes can contain only public fields, the fields must all be initialized, they cannot be static, and you cannot define any methods for them. You'll use anonymous classes periodically throughout this book and learn more about them as you do so.

## Summary

---

In this chapter, you saw how to define new classes. You learned that by default the fields and methods of a class are private and inaccessible to code outside the class, but that you can use the `public` keyword to expose fields and methods to the outside world. You saw how to use the `new` keyword to create a new instance of a class and how to define constructors that can initialize class instances. Finally, you saw how to implement static fields and methods to provide data and operations that are independent of any specific instance of a class.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 8.
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.



## Quick reference

To	Do this
Declare a class	Write the keyword <code>class</code> , followed by the name of the class, followed by opening and closing braces. The methods and fields of the class are declared between the opening and closing braces. For example: <pre>class Point {     ... }</pre>
Declare a constructor	Write a method whose name is the same as the name of the class, and that has no return type (not even <code>void</code> ). For example: <pre>class Point {     public Point(int x, int y)     {         ...     } }</pre>
Call a constructor	Use the <code>new</code> keyword and specify the constructor with an appropriate set of parameters. For example: <pre>Point origin = new Point(0, 0);</pre>
Declare a static method	Write the keyword <code>static</code> before the declaration of the method. For example: <pre>class Point {     public static int ObjectCount()     {         ...     } }</pre>
Call a static method	Write the name of the class, followed by a period, followed by the name of the method. For example: <pre>int pointsCreatedSoFar = Point.ObjectCount();</pre>
Declare a static field	Use the keyword <code>static</code> before the type of the field. For example: <pre>class Point {     ...     private static int objectCount; }</pre>
Declare a const field	Write the keyword <code>const</code> before the declaration of the field and omit the <code>static</code> keyword. For example: <pre>class Math {     ...     public const double PI = ...; }</pre>
Access a static field	Write the name of the class, followed by a period, followed by the name of the static field. For example: <pre>double area = Math.PI * radius * radius;</pre>

# Understanding values and references

**After completing this chapter, you will be able to:**

- Explain the differences between a value type and a reference type.
- Understand `null` values and how nullable types work.
- Modify how arguments are passed as method parameters by using the `ref` and `out` keywords.
- Describe how computer memory is organized to support value types and reference types.
- Convert a value into a reference by using boxing.
- Convert a reference back to a value by using unboxing and casting.

Chapter 7, “Creating and managing classes and objects,” demonstrated how to declare your own classes and how to create objects by using the `new` keyword. That chapter also showed you how to initialize an object by using a constructor. In this chapter, you’ll learn how the characteristics of the primitive types such as `int`, `double`, and `char` differ from the characteristics of class types.

## Copying value type variables and classes

---

Most of the primitive types built into C#, such as `int`, `float`, `double`, and `char` (but not `string`, for reasons that will be covered shortly), are collectively called *value types*. These types have a fixed size, and when you declare a variable as a value type, the compiler generates code that allocates a block of memory big enough to hold a corresponding value. For example, declaring an `int` variable causes the compiler to allocate 4 bytes of memory (32 bits) to hold the integer value. A statement that assigns a value (such as 42) to the `int` causes the value to be copied into this block of memory.

Class types such as `Circle` (described in Chapter 7) are handled differently. When you declare a `Circle` variable, the compiler *does not* generate code that allocates a block of memory big enough to hold a `Circle` object. All it does is allot a small piece of memory that can potentially hold the address of (or a reference to) another block of memory containing a `Circle` object. (An address specifies the location of an item in memory.) The memory for the actual `Circle` object is allocated only when the `new` keyword is used to create the object.

A class is an example of a *reference type*. Reference types hold references to blocks of memory. To write effective C# code, you must understand the difference between value types and reference types.



**Note** The `string` type in C# is actually a class. This is because there is no standard size for a string (different strings can contain different numbers of characters), and allocating memory for a string dynamically when the program runs is far more efficient than doing so statically at compile time. The description in this chapter of reference types such as classes applies to the `string` type as well. In fact, the `string` keyword in C# is just an alias for the `System.String` class.

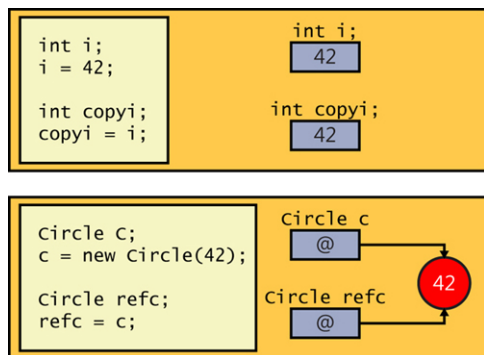
Consider a situation in which you declare a variable named `i` as an `int` and assign it the value 42. If you declare another variable called `copyi` as an `int` and then assign `i` to `copyi`, `copyi` will hold the same value as `i` (42). However, even though `copyi` and `i` happen to hold the same value, two blocks of memory contain the value 42: one block for `i` and the other block for `copyi`. If you modify the value of `i`, the value of `copyi` does not change. Let's see this in code:

```
int i = 42; // declare and initialize i
int copyi = i; /* copyi contains a copy of the data in i:
 i and copyi both contain the value 42 */
i++; /* incrementing i has no effect on copyi;
 i now contains 43, but copyi still contains 42 */
```

The effect of declaring a variable `c` as a class type, such as `Circle`, is very different. When you declare `c` as a `Circle`, `c` can refer to a `Circle` object; the actual value held by `c` is the address of a `Circle` object in memory. If you declare an additional variable named `refc` (also as a `Circle` object) and you assign `c` to `refc`, `refc` will have a copy of the same address as `c`. In other words, there's only one `Circle` object, and both `refc` and `c` now refer to it. Here's an example in code:

```
var c = new Circle(42);
Circle refc = c;
```

The following illustration shows both examples. The at sign (@) in the `Circle` objects represents a reference holding an address in memory.



This difference is very important. It means that the behavior of method parameters depends on whether they are value types or reference types. You'll explore this difference in the next exercise.

## Copying reference types and data privacy

If you actually want to copy the contents of a `Circle` object, `c`, into a different `Circle` object, `refc`, instead of just copying the reference, you must make `refc` refer to a new instance of the `Circle` class and then copy the data, field by field, from `c` into `refc`, like this:

```
var refc = new Circle();
refc.radius = c.radius; // Don't try this
```

However, if any members of the `Circle` class are private (like the `radius` field), you won't be able to copy this data. Instead, you can make the data in the private fields accessible by exposing them as properties and then use these properties to read the data from `c` and copy it into `refc`. You'll learn how to do this in Chapter 15, "Implementing properties to access fields."

Alternatively, a class could provide a `Clone` method that returns another instance of the same class but populated with the same data. The `Clone` method would have access to the private data in an object and could copy this data directly to another instance of the same class. For example, the `Clone` method for the `Circle` class could be defined as shown here:

```
class Circle
{
 private int radius;
 // Constructors and other methods omitted
 ...
 public Circle Clone()
 {
 // Create a new Circle object
 Circle clone = new Circle();

 // Copy private data from this to clone
 clone.radius = this.radius;

 // Return the new Circle object containing the copied data
 return clone;
 }
}
```

This approach is straightforward if all the private data consists of values, but if one or more fields are themselves reference types (for example, if the `Circle` class is extended to contain a `Point` object from Chapter 7, indicating the position of the `Circle` on a graph), these reference types also need to provide a `Clone` method; otherwise, the `Clone` method of the `Circle` class will simply copy a reference to these fields. This process is known as a *deep copy*. The alternative approach, wherein the `Clone` method simply copies references, is known as a *shallow copy*.

The preceding code example also poses an interesting question: How private is private data? Previously, you saw that the `private` keyword renders a field or method inaccessible from outside a class. However, this does not mean it can be accessed by only a single object. If you create two objects of the same class, they can each access the private data of the other within the code for that class.

This sounds curious, but in fact, methods such as `Clone` depend on this feature. For example, the statement `clone.radius = this.radius;` works only because the private `radius` field in the `clone` object is accessible from within the current instance of the `Circle` class. So, private actually means private to the class rather than private to an object. Don't confuse private with static, however. If you simply declare a field as private, each instance of the class gets its own data. If a field is declared as static, each instance of the class shares the same data.

## To use value parameters and reference parameters

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **Parameters** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 8\Parameters** folder in your **Documents** folder.

The project contains three C# code files: `Pass.cs`, `Program.cs`, and `WrappedInt.cs`.

3. Display the `Pass.cs` file in the Code and Text Editor window.

This file defines a class called `Pass` that is currently empty apart from a `// TODO: comment`.



**Tip** You can use the Task List window to locate all `// TODO: comment` in a solution.

4. Add a public static method called `Value` to the `Pass` class, replacing the `// TODO: comment`. This method should accept a single `int` parameter (a value type) called `param` and have the return type `void`. The body of the `Value` method should simply assign the value 42 to `param`, as shown in bold type in the following code example:

```
namespace Parameters
{
 class Pass
 {
 public static void Value(int param)
 {
 param = 42;
 }
 }
}
```



**Note** You're defining this method using the `static` keyword to keep the exercise simple. You can call the `Value` method directly on the `Pass` class without first creating a new `Pass` object. The principles illustrated in this exercise apply in the same manner to instance methods.

5. Display the Program.cs file in the Code and Text Editor window and then locate the `doWork` method of the Program class.

The `doWork` method is called by the `Main` method when the program starts running. As explained in Chapter 7, the method call is wrapped in a `try` block and followed by a catch handler.

6. Add four statements to the `doWork` method to perform the following tasks:

- Declare a local `int` variable called `i` and initialize it to 0.
- Write the value of `i` to the console by using `Console.WriteLine`.
- Call `Pass.Value`, passing `i` as an argument.
- Write the value of `i` to the console again.

By running `Console.WriteLine` before and after the call to `Pass.Value`, you can see whether the `Pass.Value` method actually modifies the value of `i`. The completed `doWork` method should look exactly like this:

```
static void doWork()
{
 int i = 0;
 Console.WriteLine(i);
 Pass.Value(i);
 Console.WriteLine(i);
}
```

7. On the **Debug** menu, select **Start Without Debugging** to build and run the program.
8. Confirm that the value 0 is written to the console window twice.

The assignment statement inside the `Pass.Value` method that updates the parameter and sets it to 42 uses a copy of the argument passed in, and the original argument `i` is completely unaffected.

9. Press **Enter** to close the application.

You'll now see what happens when you pass an `int` parameter that's wrapped within a class.

10. Display the `WrappedInt.cs` file in the Code and Text Editor window. This file contains the `WrappedInt` class, which is empty apart from a `// TODO:`  comment.
11. Add a public instance field called `Number` of type `int` to the `WrappedInt` class, as shown in bold type in the following code:

```
namespace Parameters
{
 class WrappedInt
 {
 public int Number;
 }
}
```

12. Display the `Pass.cs` file in the Code and Text Editor window. Add a public static method called `Reference` to the `Pass` class. This method should accept a single `WrappedInt` parameter called `param` and have the return type `void`. The body of the `Reference` method should assign 42 to `param.Number`, as shown here:

```
public static void Reference(WrappedInt param)
{
 param.Number = 42;
}
```

13. Display the `Program.cs` file in the Code and Text Editor window. Comment out the existing code in the `doWork` method and add four more statements to perform the following tasks:

- Declare a local `WrappedInt` variable called `wi` and initialize it to a new `WrappedInt` object by calling the default constructor.
- Write the value of `wi.Number` to the console.
- Call the `Pass.Reference` method, passing `wi` as an argument.
- Write the value of `wi.Number` to the console again.

As before, with the calls to `Console.WriteLine`, you can see whether the call to `Pass.Reference` modifies the value of `wi.Number`. The `doWork` method should now look exactly like this (the new statements are in bold):

```
static void doWork()
{
 // int i = 0;
 // Console.WriteLine(i);
 // Pass.Value(i);
 // Console.WriteLine(i);
 var wi = new WrappedInt();
 Console.WriteLine(wi.Number);
 Pass.Reference(wi);
 Console.WriteLine(wi.Number);
}
```

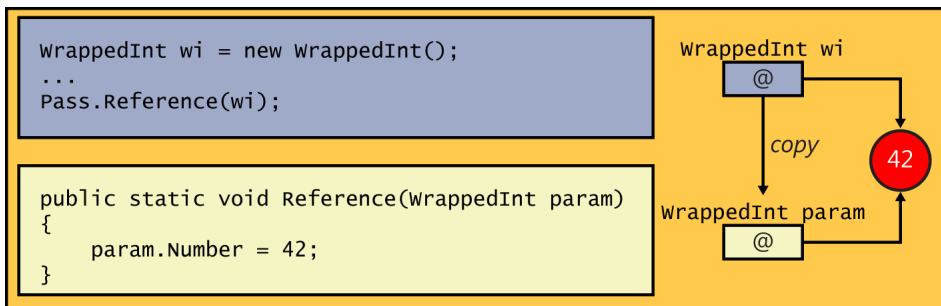
14. On the **Debug** menu, select **Start Without Debugging** to build and run the application.

This time, the two values displayed in the console window correspond to the value of `wi.Number` before and after the call to the `Pass.Reference` method. You should see that the values 0 and 42 are displayed.

15. Press **Enter** to close the application and return to Visual Studio 2022.

To explain what the previous exercise shows, the value of `wi.Number` is initialized to 0 by the compiler-generated default constructor. The `wi` variable contains a reference to the newly created `WrappedInt` object (which contains an `int`). The `wi` variable is then copied as an argument to the `Pass.Reference` method. Because `WrappedInt` is a class (a reference type), `wi` and `param` both refer to the same `WrappedInt` object. Any changes made to the contents of the object through the `param` variable in the `Pass.Reference` method are visible by using the `wi` variable when the method

completes. The following diagram illustrates what happens when a `WrappedInt` object is passed as an argument to the `Pass.Reference` method.



## Understanding null values and nullable types

When you declare a variable, it's always a good idea to initialize it. With value types, it's common to see code such as this:

```
int i = 0;
double d = 0.0;
```

To initialize a reference variable such as a class, you can create a new instance of the class and assign the reference variable to the new object, like this:

```
var c = new Circle(42);
```

This is all very well, but what if you don't actually want to create a new object? Perhaps the purpose of the variable is simply to store a reference to an existing object at some later point in your program. In the following code example, the `Circle` variable `copy` is initialized, but later it is assigned a reference to another instance of the `Circle` class:

```
var c = new Circle(42);
var copy = new Circle(99); // Some random value for initializing copy
...
copy = c; // copy and c refer to the same object
```

After assigning `c` to `copy`, what happens to the original `Circle` object with a radius of 99 that you used to initialize `copy`? Nothing refers to it anymore. In this situation, the runtime can reclaim the memory by performing an operation known as *garbage collection*, which you'll learn more about in Chapter 14, "Using garbage collection and resource management." The important thing to understand for now is that garbage collection is a potentially time-consuming operation, and that you should not create objects that are never used because doing so is a waste of time and resources.

You could argue that if a variable will be assigned a reference to another object at some point in a program, there's no point to initializing it. This is poor programming practice, however, and can lead to problems in your code. For example, you will inevitably find yourself in a situation in which you want to



refer a variable to an object only if that variable does not already contain a reference, as shown in the following code example:

```
var c = new Circle(42);
Circle copy; // Uninitialized !!!
...
if (copy == // only assign to copy if it is uninitialized, but what goes here?)
{
 copy = c; ; // copy and c refer to the same object
 ...
}
```

The purpose of the `if` statement is to test the `copy` variable to see whether it is initialized, but to which value should you compare this variable? The answer is to use a special value called `null`.

In C#, you can assign the `null` value to any reference variable. The `null` value simply means that the variable does not refer to an object in memory. You can use it like this:

```
Circle c = new Circle(42);
Circle copy = null; // Initialized
...
if (copy is null)
{
 copy = c; // copy and c refer to the same object
 ...
}
```



**Note** You can also use `== null` to check for a null reference. However, `is null` reads more naturally. Similarly, you can use `is not null` as well as `!= null` to check for a non-null reference.

## The null-conditional and null-coalescing operators

The null-conditional operator enables you to test for `null` values very succinctly. To use the null-conditional operator, you append a question mark (?) to the name of your variable.

For example, suppose you attempt to call the `Area` method on a `Circle` object when the `Circle` object has a `null` value:

```
Circle c = null;
Console.WriteLine($"The area of circle c is {c.Area()}");
```

In this case, the `Circle.Area` method throws a `NullReferenceException`, which makes sense because you cannot calculate the area of a circle that does not exist.

To avoid this exception, you could test whether the `Circle` object is `null` before you attempt to call the `Circle.Area` method:

```
if (c is not null)
{
 Console.WriteLine($"The area of circle c is {c.Area()}");
}
```

In this case, if `c` is `null`, nothing is written to the command window. Alternatively, you could use the null-conditional operator on the `Circle` object before you attempt to call the `Circle.Area` method:

```
Console.WriteLine($"The area of circle c is {c?.Area()}");
```

The null-conditional operator tells the C# runtime to ignore the current statement if the variable you have applied the operator to is `null`. In this case, the command window would display the following text:

```
The area of circle c is
```

Both approaches are valid and might meet your needs in different scenarios. The null-conditional operator can help you keep your code concise, particularly when you deal with complex properties with nested reference types that could all be `null` valued.

Alongside the null-conditional operator, C# provides two null-coalescing operators. The first of these, `??`, is a binary operator that returns the value of the operand on the left if it isn't `null`; otherwise, it returns the value of the operand on the right. In the following example, variable `c2` is assigned a reference to `c` if `c` isn't `null`; otherwise, it is assigned a reference to a new `Circle` object:

```
Circle c = ...; // might be null, might be a new Circle object
...
var c2 = c ?? new Circle(42);
```

The null-coalescing assignment operator, `??=`, assigns the value of the operand on the right to the operand on the left only if the left operand is `null`. If the left operand references some other value, it is unchanged.

```
Circle c = ...; // might be null, might be a new Circle object
Circle c3 = ...; // might be null, might be a new Circle object
...
var c3 ??= c; // Only assign c3 if it is null, otherwise leave unchanged;
```

## Using nullable types

The `null` value is very useful for initializing reference types. Sometimes, though, you need an equivalent value for value types. `null` is itself a reference, so you cannot assign it to a value type. The following statement is therefore illegal in C#:

```
int i = null; // illegal
```

However, C# defines a modifier that you can use to declare that a variable is a *nullable* value type. A nullable value type behaves similarly to the original value type, but you can assign the `null` value to it. You use the question mark (`?`) to indicate that a value type is nullable, like this:

```
int? i = null; // legal
```

You can ascertain whether a nullable variable contains `null` by testing it in the same way as you test a reference type.

```
if (i is null)
 ...
```

You can assign an expression of the appropriate value type directly to a nullable variable. The following examples are all legal:

```
int? i = null;
int j = 99;
i = 100; // Copy a value type constant to a nullable type
i = j; // Copy a value type variable to a nullable type
```

You should note that the converse is not true. You cannot assign a nullable variable to an ordinary value type variable. So, given the definitions of variables `i` and `j` from the preceding example, the following statement is not allowed:

```
j = i; // illegal
```

This makes sense when you consider that the variable `i` might contain `null`, and `j` is a value type that cannot contain `null`. This also means that you cannot use a nullable variable as a parameter to a method that expects an ordinary value type. If you recall, the `Pass.Value` method from the preceding exercise expects an ordinary `int` parameter, so the following method call will not compile:

```
int? i = 99;
Pass.Value(i); // Compiler error
```



**Note** Take care not to confuse nullable types with the null-conditional operator. Nullable types are indicated by appending a question mark to the type name, whereas the null-conditional operator is appended to the variable name.

## Understanding the properties of nullable types

A nullable type exposes a pair of properties that you can use to determine whether the type actually has a non-null value and what this value is. The `HasValue` property indicates whether a nullable type contains a value or is `null`. You can retrieve the value of a non-null nullable type by reading the `Value` property, like this:

```
int? i = null;
...
if (!i.HasValue)
{
 // If i is null, then assign it the value 99
 i = 99;
}
else
{
 // If i is not null, then display its value
 Console.WriteLine(i.Value);
}
```

In Chapter 4, “Using decision statements,” you saw that the NOT operator (!) negates a Boolean value. The preceding code fragment tests the nullable variable `i`, and if it does not have a value (it is `null`), it assigns it the value 99; otherwise, it displays the value of the variable. In this example, using the `HasValue` property does not provide any benefit over testing for a `null` value directly. Additionally, reading the `Value` property is a long-winded way of reading the contents of the variable. However, these apparent shortcomings are caused by the fact that `int?` is a very simple nullable type. You can create more complex value types and use them to declare nullable variables where the advantages of using the `HasValue` and `Value` properties become more apparent. You’ll see some examples in Chapter 9, “Creating value types with enumerations and structures.”



**Note** The `Value` property of a nullable type is read-only. You can use this property to read the value of a variable but not to modify it. To update a nullable variable, use an ordinary assignment statement.

## Using ref and out parameters

Ordinarily, when you pass an argument to a method, the corresponding parameter is initialized with a copy of the argument. This is true regardless of whether the parameter is a value type (such as an `int`), a nullable type (such as `int?`), or a reference type (such as a `WrappedInt`). This arrangement means that it’s impossible for any change to the parameter to affect the value of the argument passed in. For example, in the following code, the value output to the console is 42, not 43. The `doIncrement` method increments a copy of the argument (`arg`) and *not* the original argument, as demonstrated here:

```
static void doIncrement(int param)
{
 param++;
}

static void Main()
{
 int arg = 42;
 doIncrement(arg);
 Console.WriteLine(arg); // writes 42, not 43
}
```

In the preceding exercise, you saw that if the parameter to a method is a reference type, any changes made by using that parameter change the data referenced by the argument passed in. The key point is this: although the data that was referenced changed, the argument passed in as the parameter did not. It still references the same object. In other words, although it’s possible to modify the object that the argument refers to through the parameter, it’s not possible to modify the argument itself—for example, to set it to refer to a completely different object. Most of the time, this guarantee is very useful and can help reduce the number of bugs in a program. Occasionally, however, you might want to write a method that actually needs to modify an argument. C# provides the `ref` and `out` keywords so that you can do this.

## Creating ref parameters

If you prefix a parameter with the `ref` keyword, the C# compiler generates code that passes a reference to the actual argument rather than a copy of the argument. When using a `ref` parameter, anything you do to the parameter you also do to the original argument because the parameter and the argument both reference the same data.

When you pass an argument as a `ref` parameter, you must also prefix the argument with the `ref` keyword. This syntax provides a useful visual cue to the programmer that the argument might change. Here's the preceding example again, this time modified to use the `ref` keyword:

```
static void doIncrement(ref int param) // using ref
{
 param++;
}

static void Main()
{
 int arg = 42;
 doIncrement(ref arg); // using ref
 Console.WriteLine(arg); // writes 43
}
```

This time, the `doIncrement` method receives a reference to the original argument rather than a copy, so any changes the method makes by using this reference actually change the original value. That's why the value 43 is displayed on the console.

Remember that C# enforces the rule that you must assign a value to a variable before you can read it. This rule also applies to method arguments; you cannot pass an uninitialized value as an argument to a method even if an argument is defined as a `ref` argument. For example, in the following example, `arg` is not initialized, so this code will not compile. This failure occurs because the statement `param++`; within the `doIncrement` method is really an alias for the statement `arg++`; and this operation is allowed only if `arg` has a defined value:

```
static void doIncrement(ref int param)
{
 param++;
}

static void Main()
{
 int arg; // not initialized
 doIncrement(ref arg);
 Console.WriteLine(arg);
}
```

## Creating out parameters

The compiler checks whether a `ref` parameter has been assigned a value before calling the method. However, there might be times when you want the method itself to initialize the parameter. You can do this with the `out` keyword.

The `out` keyword is syntactically similar to the `ref` keyword. You can prefix a parameter with the `out` keyword so that the parameter becomes an alias for the argument. As when using `ref`, anything you do to the parameter, you also do to the original argument. When you pass an argument to an `out` parameter, you must also prefix the argument with the `out` keyword.

The keyword `out` is short for *output*. When you pass an `out` parameter to a method, the method *must* assign a value to it before it finishes or returns, as shown in the following example:

```
static void doInitialize(out int param)
{
 param = 42; // Initialize param before finishing
}
```

The following example does not compile because `doInitialize` does not assign a value to `param`:

```
static void doInitialize(out int param)
{
 // Do nothing
}
```

Because an `out` parameter must be assigned a value by the method, you're allowed to call the method without initializing its argument. For example, the following code calls `doInitialize` to initialize the variable `arg`, which is then displayed on the console:

```
static void doInitialize(out int param)
{
 param = 42;
}

static void Main()
{
 int arg; // not initialized
 doInitialize(out arg); // legal
 Console.WriteLine(arg); // writes 42
}
```



**Note** You can combine the declaration of an `out` variable with its use as a parameter rather than performing these tasks separately. For example, you could replace the first two statements in the `Main` method in the previous example with this single line of code:

```
doInitialize(out int arg);
```

In the next exercise, you'll practice using `ref` parameters.

### To use `ref` parameters

1. Return to the Parameters project in Visual Studio 2022.
2. Display the `Pass.cs` file in the Code and Text Editor window.

3. Edit the `Value` method to accept its parameter as a `ref` parameter.

The `Value` method should look like this:

```
class Pass
{
 public static void Value(ref int param)
 {
 param = 42;
 }
 ...
}
```

4. Display the `Program.cs` file in the Code and Text Editor window.
5. Uncomment the first four statements.

Notice that the third statement of the `doWork` method, `Pass.Value(i)`, indicates an error. The error occurs because the `Value` method now expects a `ref` parameter.

6. Edit this statement so that the `Pass.Value` method call passes its argument as a `ref` parameter.



**Note** Leave the four statements that create and test the `WrappedInt` object as they are.

The `doWork` method should now look like this:

```
class Program
{
 static void doWork()
 {
 int i = 0;
 Console.WriteLine(i);
 Pass.Value(ref i);
 Console.WriteLine(i);
 ...
 }
}
```

7. On the **Debug** menu, select **Start Without Debugging** to build and run the program.

This time, the first two values written to the console window are 0 and 42. This result shows that the call to the `Pass.Value` method has successfully modified the argument `i`.

8. Press **Enter** to close the application and return to Visual Studio 2022.



**Note** You can use the `ref` and `out` modifiers on reference type parameters as well as on value type parameters. The effect is the same: the parameter becomes an alias for the argument.

# How computer memory is organized

Computers use memory to hold programs that are being executed and the data that those programs use. To understand the differences between value and reference types, it's helpful to understand how data is organized in memory.

Operating systems and language runtimes such as those used by C# frequently divide the memory used for holding data into two separate areas, each of which is managed in a distinct manner. These two areas of memory are traditionally called the *stack* and the *heap*. The stack and the heap serve different purposes:

- When you call a method, the memory required for its parameters and its local variables is acquired from the stack. When the method finishes (because it either returns or throws an exception), the memory acquired for the parameters and local variables is automatically released back to the stack to be made available again when another method is called. Method parameters and local variables on the stack have a well-defined lifespan: They come into existence when the method starts, and they disappear as soon as the method completes.

The same lifespan applies to variables defined in any block of code enclosed by opening and closing braces. In the following code example, the variable `i` is created when the body of the `while` loop starts, but it disappears when the `while` loop finishes, and execution continues after the closing brace:

```
while (...)
{
 int i = ...; // i is created on the stack here
 ...
}
// i disappears from the stack here
```

- When you create an object (an instance of a class) by using the `new` keyword, the memory required to build the object is acquired from the heap. You've seen that the same object can be referenced from several places by using reference variables. When the last reference to an object disappears, the memory used by the object becomes available again (although it might not be reclaimed immediately). Objects created on the heap therefore have a more indeterminate lifespan; an object is created by using the `new` keyword, but it disappears only sometime after the last reference to the object is removed. Chapter 14 includes a more detailed discussion of how heap memory is reclaimed.



**Note** All value types are created on the stack. By default, reference types (objects) are created on the heap, although the reference itself is on the stack. (There are some exceptions to this rule, which you'll learn about in later chapters.) Nullable objects are actually reference types, and they are created on the heap.



The names *stack* and *heap* come from the way in which the runtime manages the memory:

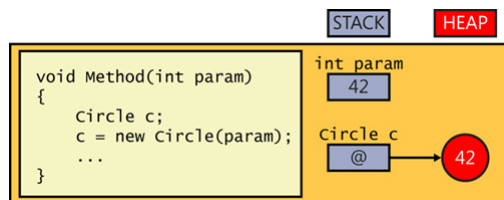
- Stack memory is organized like boxes stacked neatly on top of one another. When a method is called, each parameter is placed in a box that is added to the top of the stack. Each local variable is likewise assigned a box, which is placed on top of the boxes already on the stack. When a method finishes, think of it as being like a box being removed from the stack.
- Heap memory is like a large pile of boxes strewn around a room rather than stacked neatly on top of one another. Each box has a label indicating whether it is in use. When a new object is created, the runtime searches for an empty box and allocates it to the object. The reference to the object is stored in a local variable on the stack. The runtime keeps track of the number of references to each box. (Remember: two variables can refer to the same object.) When the last reference disappears, the runtime marks the box as not in use; at some point in the future, it empties the box and makes it available.

## Using the stack and the heap

Now let's examine what happens when a method named `Method` is called:

```
void Method(int param)
{
 Circle c;
 c = new Circle(param);
 ...
}
```

Suppose the argument passed into `param` is the value 42. When the method is called, a block of memory (just enough for an `int`) is allocated from the stack and initialized with the value 42. As execution moves inside the method, another block of memory big enough to hold a reference (a memory address) is also allocated from the stack, but left uninitialized. This is for the `Circle` variable, `c`. Next, another piece of memory big enough for a `Circle` object is allocated from the heap. This is what the `new` keyword does. The `Circle` constructor runs to convert this raw heap memory to a `Circle` object. A reference to this `Circle` object is stored in the variable `c`. The following illustration shows this process:



At this point, you should note two things:

- Although the object is stored on the heap, the reference to the object (the variable `c`) is stored on the stack.
- Heap memory is not infinite. If heap memory is exhausted, the `new` operator will throw an `OutOfMemoryException` exception, and the object will not be created.



**Note** The `Circle` constructor could also throw an exception. If it does, the memory allocated to the `Circle` object will be reclaimed, and the value returned by the constructor will be `null`.

When the method ends, the parameters and local variables go out of scope. The memory acquired for `c` and `param` is automatically released back to the stack. The runtime notes that the `Circle` object is no longer referenced and at some point in the future will arrange for its memory to be reclaimed by the heap. (See Chapter 14.)

## The `System.Object` class

One of the most important reference types in .NET is the `Object` class in the `System` namespace. To fully appreciate the significance of the `System.Object` class, you must understand inheritance, which is described in Chapter 12, “Working with inheritance.” For now, simply accept that all classes are specialized types of `System.Object` and that you can use `System.Object` to create a variable that can refer to any reference type. `System.Object` is such an important class that C# provides the `object` keyword as an alias for `System.Object`. In your code, you can use `object`, or you can write `System.Object`. They mean the same thing.

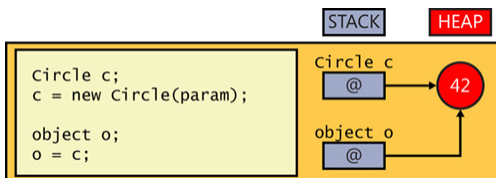


**Tip** Use the `object` keyword rather than `System.Object`. It’s more direct, and it’s consistent with other keywords that are synonyms for classes, such as `string` for `System.String` and others that are covered in Chapter 9.

In the following example, the variables `c` and `o` both refer to the same `Circle` object. The fact that the type of `c` is `Circle` and the type of `o` is `object` (the alias for `System.Object`) in effect provides two different views of the same item in memory.

```
Circle c;
c = new Circle(42);
object o;
o = c;
```

The following diagram illustrates how the variables `c` and `o` refer to the same item on the heap:

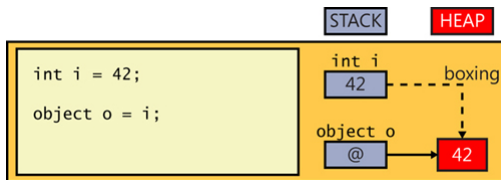


## Boxing

As you have just seen, variables of type `object` can refer to any item of any reference type. However, variables of type `object` can also refer to a value type. For example, the following two statements initialize the variable `i` (of type `int`, a value type) to 42 and then initialize the variable `o` (of type `object`, a reference type) to `i`:

```
int i = 42;
object o = i;
```

The second statement requires a little explanation to appreciate what's actually happening. Remember that `i` is a value type and that it lives on the stack. If the reference inside `o` referred directly to `i`, the reference would refer to the stack. However, references should refer to objects on the heap. Creating uncontrolled references to items on the stack could seriously compromise the robustness of the runtime and potentially create a security flaw, so it is not allowed. Therefore, the runtime allocates a piece of memory from the heap, copies the value of integer `i` to this piece of memory, and then refers the object `o` to this copy. This automatic copying of an item from the stack to the heap is called *boxing*. The following diagram shows the result:



**Important** If you modify the original value of the variable `i`, the value on the heap referenced through `o` will not change. Likewise, if you modify the value on the heap, the original value of the variable will not change.

## Unboxing

Because a variable of type `object` can refer to a boxed copy of a value, it's only reasonable to allow you to get at that boxed value through the variable. You might expect to be able to access the boxed `int` value that a variable `o` refers to by using a simple assignment statement such as this:

```
int i = o;
```

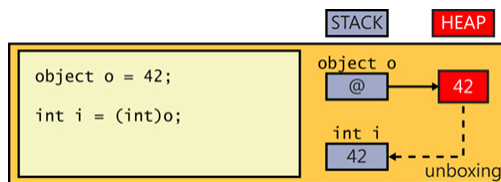
However, if you try this syntax, you'll get a compile-time error. If you think about it, it's fairly sensible that you can't use the `int i = o;` syntax. After all, `o` could be referencing absolutely anything and not just an `int`. Consider what would happen in the following code if this statement were allowed:

```
Circle c = new Circle();
int i = 42;
object o;
o = c; // o refers to a circle
i = o; // what is stored in i?
```

To obtain the value of the boxed copy, you must use what is known as a *cast*. This is an operation that checks whether converting an item of one type to another is safe before actually making the copy. You prefix the object variable with the name of the type in parentheses, as in this example:

```
int i = 42;
object o = i; // boxes
i = (int)o; // compiles okay
```

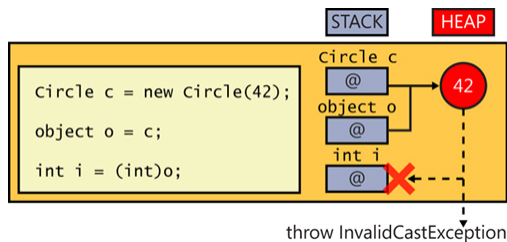
The effect of this cast is subtle. The compiler notices that you've specified the type `int` in the cast. Next, the compiler generates code to check what `o` actually refers to at runtime. It could be absolutely anything. Just because your cast says `o` refers to an `int`, that doesn't mean it actually does. If `o` really does refer to a boxed `int` and everything matches, the cast succeeds, and the compiler-generated code extracts the value from the boxed `int` and copies it to `i`. (In this example, the boxed value is then stored in `i`.) This is called *unboxing*. The following diagram shows what's happening:



On the other hand, if `o` does not refer to a boxed `int`, there is a type mismatch, causing the cast to fail. The compiler-generated code throws an `InvalidCastException` exception at runtime. Here's an example of an unboxing cast that fails:

```
Circle c = new Circle(42);
object o = c; // doesn't box because Circle is a reference variable
int i = (int)o; // compiles okay but throws an exception at runtime
```

The following diagram illustrates this case:



You'll use boxing and unboxing in later exercises. Keep in mind that boxing and unboxing are expensive operations because of the amount of checking required and the need to allocate additional heap memory. Boxing has its uses, but injudicious use can severely impair the performance of a program. You'll see an alternative to boxing in Chapter 17, "Introducing generics."

## Casting data safely

---

By using a cast, you can specify that, in your opinion, the data referenced by an object has a specific type and that it's safe to reference the object by using that type. The key phrase here is “in your opinion.” The C# compiler will not check that this is the case, but the runtime will. If the type of object in memory does not match the cast, the runtime will throw an `InvalidCastException`, as described in the preceding section. You should be prepared to catch this exception and handle it appropriately if it occurs.

However, catching an exception and attempting to recover if the type of an object is not what you expected it to be is a rather cumbersome approach. C# provides two more very useful operators that can help you perform casting in a much more elegant manner: the `is` and `as` operators.

### The `is` operator

You've seen the `is` operator before, when checking for a `null` value, but it actually enables you to check for the type of any reference object. You can use the `is` operator to verify that the type of an object is what you expect it to be, like this:

```
var wi = new WrappedInt();
...
object o = wi;
if (o is WrappedInt)
{
 WrappedInt temp = (WrappedInt)o; // This is safe; o is a WrappedInt
 ...
}
```

The `is` operator takes two operands: a reference to an object on the left, and the name of a type (or `null`) on the right. If the type of the object referenced on the heap matches the type specified by the `is` operator, `is` evaluates to `true`; otherwise, `is` evaluates to `false`. The preceding code attempts to cast the reference to the object variable `o` only if it knows that the cast will succeed.

Another form of the `is` operator enables you to abbreviate this code by combining the type check and the assignment, like this:

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
...
if (o is WrappedInt temp)
{
 ... // Use temp here
}
```

In this example, if the test for the `WrappedInt` type is successful, the `is` operator creates a new reference variable (called `temp`) and assigns it a reference to the `WrappedInt` object.

## The as operator

The `as` operator fulfills a similar role to `is` but in a slightly truncated manner. You use the `as` operator like this:

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
WrappedInt temp = o as WrappedInt;
if (temp is not null)
{
 ... // Cast was successful
}
```

Like the `is` operator, the `as` operator takes an object and a type as its operands. The runtime attempts to cast the object to the specified type. If the cast is successful, the result is returned and, in this example, is assigned to the `WrappedInt` variable `temp`. If the cast is unsuccessful, the `as` operator evaluates to the `null` value and assigns that to `temp` instead.



**Note** There's a little more to the `is` and `as` operators than is described here; Chapter 12 discusses them in greater detail.

## The switch statement revisited

If you need to check a reference against several types, you can use a series of `if...else` statements in conjunction with the `is` operator. The following example assumes that you have defined the `Circle`, `Square`, and `Triangle` classes. The constructors take the radius (`radius`) or side length (`side`) of the geometric shape as the parameter:

```
var c = new Circle(42); // Circle of radius 42
var s = new Square(55); // Square of side 55
var t = new Triangle(33); // Equilateral triangle of side 33
...
object o = s;
...
if (o is Circle myCircle)
{
 ... // o is a Circle, a reference is available in myCircle
}
else if (o is Square mySquare)
{
 ... // o is a Square, a reference is available in mySquare
}
else if (o is Triangle myTriangle)
{
 ... // o is a Triangle, a reference is available in myTriangle
}
```

As with any lengthy set of `if...else` statements, this approach can quickly become cumbersome and difficult to read. Fortunately, you can use the `switch` statement in this situation, as follows:

```
switch (o)
{
 case Circle myCircle:
 ... // o is a Circle, a reference is available in myCircle
 break;

 case Square mySquare:
 ... // o is a Square, a reference is available in mySquare
 break;

 case Triangle myTriangle:
 ... // o is a Triangle, a reference is available in myTriangle
 break;

 default:
 throw new ArgumentException("variable is not a recognized shape");
 break;
}
```

In both examples (using the `is` operator and the `switch` statement), the scope of the variables created (`myCircle`, `mySquare`, and `myTriangle`) is limited to the code inside the corresponding `if` block or `case` block.

case selectors in `switch` statements also support `when` expressions, which you can use to further qualify the situation under which the case is selected. For example, the following `switch` statement shows case selectors that match different sizes of geometric shapes:

```
switch (o)
{
 case Circle myCircle when myCircle.Radius > 10:
 ...
 break;
 case Square mySquare when mySquare.SideLength == 100:
 ...
 break;
 ...
}
```

## Pointers and unsafe code

This sidebar is purely for your information and is aimed at developers who are familiar with C or C++. If you're new to programming, feel free to ignore this information.

If you have already written programs in languages such as C or C++, much of the discussion in this chapter concerning object references might be familiar in that both languages have a construct that provides similar functionality: a pointer. A *pointer* is a variable that holds the address of, or a reference to, an item in memory (on the heap or the stack).

A special syntax is used to identify a variable as a pointer. For example, the following statement declares the variable `pi` as a pointer to an integer:

```
int *pi;
```

Although the variable `pi` is declared as a pointer, it does not actually point anywhere until you initialize it. For example, to use `pi` to point to the integer variable `i`, you can use the following statements and the address-of operator (`&`), which returns the address of a variable:

```
int *pi;
int i = 99;
...
pi = &i;
```

You can access and modify the value held in the variable `i` through the pointer variable `pi` like this:

```
*pi = 100;
```

This code updates the value of the variable `i` to 100 because `pi` points to the same memory location as the variable `i`.

One of the main problems that developers learning C and C++ encounter is understanding the syntax used by pointers. The `*` operator has at least two meanings (in addition to being the arithmetic multiplication operator), and there's often great confusion about when to use `&` rather than `*`.

The other issue with pointers is that it's easy to point somewhere invalid or to forget to point somewhere at all, and then try to reference the data pointed to. The result will be either garbage or a program that fails with an error because the operating system detects an attempt to access an illegal address in memory.

Finally, there are several security flaws in many existing systems resulting from the mismanagement of pointers. Some environments (not Windows) fail to enforce checks that a pointer does not refer to memory that belongs to another process, opening up the possibility that confidential data could be compromised.

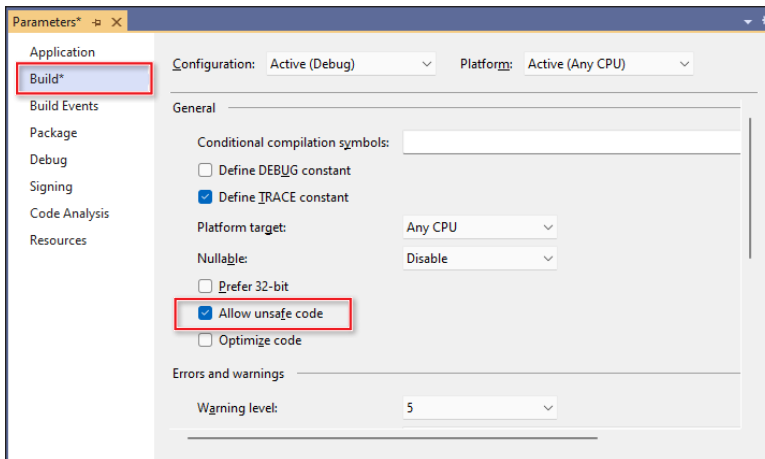


Reference variables were added to C# to avoid all these problems. If you really want to, you can continue to use pointers in C#, but you must mark the code as unsafe. The unsafe keyword can be used to mark a block of code or an entire method, as shown here:

```
public static void Main(string [] args)
{
 int x = 99, y = 100;
 unsafe
 {
 swap (&x, &y);
 }
 Console.WriteLine($"x is now {x}, y is now {y}");
}

public static unsafe void swap(int *a, int *b)
{
 int temp;
 temp = *a;
 *a = *b;
 *b = temp;
}
```

When you compile programs containing unsafe code, you must specify the Allow Unsafe Code option when building the project. To do this, right-click the project in Solution Explorer and then select Properties. In the Properties window, select the Build tab, select Allow Unsafe Code, and then, on the File menu, select Save All.



Unsafe code also affects how memory is managed. Objects created in unsafe code are said to be unmanaged. Although situations that require you to access memory in this way are not common, you might encounter some, especially if you're writing code that needs to perform some low-level Windows operations.

You'll learn about the implications of using code that accesses unmanaged memory in more detail in Chapter 14.

# Summary

In this chapter, you learned about some important differences between value types that hold their value directly on the stack and reference types that refer indirectly to their objects on the heap. You also learned how to use the `ref` and `out` keywords on method parameters to gain access to the arguments. You saw how assigning a value (such as the `int 42`) to a variable of the `System.Object` class creates a boxed copy of the value on the heap and then causes the `System.Object` variable to refer to this boxed copy. You also saw how assigning a variable of a value type (such as an `int`) from a variable of the `System.Object` class copies (or unboxes) the value in the `System.Object` class to the memory used by the `int`.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 9.
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

## Quick reference

To	Do this
Copy a value type variable	Simply make the copy. Because the variable is a value type, you will have two copies of the same value. For example: <pre>int i = 42; int copyi = i;</pre>
Copy a reference type variable	Simply make the copy. Because the variable is a reference type, you will have two references to the same object. For example: <pre>Circle c = new Circle(42); Circle refc = c;</pre>
Declare a variable that can hold a value type or the <code>null</code> value	Declare the variable by using the <code>?</code> modifier with the type. For example: <pre>int? i = null;</pre>
Pass an argument to a <code>ref</code> parameter	Prefix the argument with the <code>ref</code> keyword. This makes the parameter an alias for the actual argument rather than a copy of the argument. The method may change the value of the parameter, and this change is made to the actual argument rather than to a local copy. For example: <pre>static void Main() {     int arg = 42;     doWork(ref arg);     Console.WriteLine(arg); }</pre>
Pass an argument to an <code>out</code> parameter	Prefix the argument with the <code>out</code> keyword. This makes the parameter an alias for the actual argument rather than a copy of the argument. The method must assign a value to the parameter, and this value is made to the actual argument. For example: <pre>static void Main() {     int arg;     doWork(out arg);     Console.WriteLine(arg); }</pre>

To	Do this
Box a value	Initialize or assign a variable of type object with the value. For example: <pre>object o = 42;</pre>
Unbox a value	Cast the object reference that refers to the boxed value to the type of the value variable. For example: <pre>int i = (int)o;</pre>
Cast an object safely	Use the is operator to test whether the cast is valid. For example: <pre>WrappedInt wi = new WrappedInt(); ... object o = wi; if (o is WrappedInt temp) {     ... }</pre> Alternatively, use the as operator to perform the cast, and test whether the result is null. For example: <pre>WrappedInt wi = new WrappedInt(); ... object o = wi; WrappedInt temp = o as WrappedInt; if (temp != null)     ...</pre>

# Creating value types with enumerations and structures

**After completing this chapter, you will be able to:**

- Declare an enumeration type and create an enumeration variable.
- Declare a structure type and create a structure variable.
- Explain the differences in behavior between a structure and a class.

Chapter 8, “Understanding values and references,” covers the two fundamental types that exist in Microsoft Visual C#: value types and reference types. Recall that a value type variable holds its value directly on the stack, whereas a reference type variable holds a reference to an object on the heap. Chapter 7, “Creating and managing classes and objects,” demonstrated how to create your own reference types by defining classes. In this chapter, you’ll learn how to create your own value types.

C# supports two kinds of value types: *enumerations* and *structures*. We’ll look at each of them in turn.

## Working with enumerations

---

Suppose you want to represent the seasons of the year in a program. You could use the integers 0, 1, 2, and 3 to represent spring, summer, fall, and winter, respectively. This system would work, but it’s not very intuitive. If you used the integer value 0 in code, it wouldn’t be obvious that a particular 0 represented spring. It also wouldn’t be a very robust solution. For example, if you declare an `int` variable named `season`, there’s nothing to stop you from assigning it any legal integer value outside the set 0, 1, 2, or 3. C# offers a better solution: you can create an enumeration (sometimes called an `enum` type) whose values are limited to a set of symbolic names.

### Declaring an enumeration

You define an enumeration by using the `enum` keyword followed by a set of symbols identifying the legal values that the type can have enclosed within braces. Here’s how you declare an enumeration named `Season` whose literal values are limited to the names `Spring`, `Summer`, `Fall`, and `Winter`:

```
enum Season { Spring, Summer, Fall, Winter }
```

## Using an enumeration

After you've declared an enumeration, you can use it in the same way you do any other type. If the name of your enumeration is `Season`, you can create variables of type `Season`, fields of type `Season`, and parameters of type `Season`, as shown in this example:

```
enum Season { Spring, Summer, Fall, Winter }
class Example
{
 public void Method(Season parameter) // method parameter example
 {
 Season localVariable; // local variable example
 ...
 }

 private Season currentSeason; // field example
}
```

Before you can read the value of an enumeration variable, it must be assigned a value. You can assign a value that is defined by the enumeration only to an enumeration variable, as illustrated here:

```
Season colorful = Season.Fall;
Console.WriteLine(colorful); // writes out 'Fall'
```



**Note** As with all value types, you can create a nullable version of an enumeration variable by using the `?` modifier. You can then assign the `null` value, as well as the values defined by the enumeration, to the variable:

```
Season? colorful = null;
```

Notice that you must write `Season.Fall` rather than just `Fall`. All enumeration literal names are scoped by their enumeration type, which makes it possible for different enumerations to contain literals with the same name.

Also, notice that when you display an enumeration variable by using `Console.WriteLine`, the compiler generates code that writes out the name of the literal whose value matches the value of the variable. If needed, you can explicitly convert an enumeration variable to a string that represents its current value by using the built-in `ToString` method that all enumerations automatically contain, as demonstrated in the following example:

```
string name = colorful.ToString();
Console.WriteLine(name); // also writes out 'Fall'
```

You can employ many of the standard operators used for integer variables on enumeration variables (except the `bitwise` and `shift` operators, which are covered in Chapter 16, "Handling binary data and using indexers"). For example, you can compare two enumeration variables of the same type for equality by using the equality operator (`==`), and you can even perform arithmetic on an enumeration variable—although the result might not always be meaningful!

## Choosing enumeration literal values

Internally, an enumeration type associates an integer value with each element of the enumeration. By default, the numbering starts at 0 for the first element and goes up in steps of 1.

It's possible to retrieve the underlying integer value of an enumeration variable. To do this, you must cast it to its underlying type. The discussion in Chapter 8 on unboxing explains that casting a type converts the data from one type to another as long as the conversion is valid and meaningful. The following code example writes out the value 2 and not the word `Fall` (remember, in the `Season` enumeration, `Spring` is 0, `Summer` 1, `Fall` 2, and `Winter` 3):

```
enum Season { Spring, Summer, Fall, Winter }
...
Season colorful = Season.Fall;
Console.WriteLine((int)colorful); // writes out '2'
```

If you prefer, you can associate a specific integer constant (such as 1) with an enumeration literal (such as `Spring`), as in the following example:

```
enum Season { Spring = 1, Summer, Fall, Winter }
```



**Important** The integer value with which you initialize an enumeration literal must be a compile-time constant value (such as 1).

If you don't explicitly give an enumeration literal a constant integer value, the compiler gives it a value that is one greater than the value of the previous enumeration literal, except for the very first enumeration literal, to which the compiler gives the default value 0. In the preceding example, the underlying values of `Spring`, `Summer`, `Fall`, and `Winter` are now 1, 2, 3, and 4.

You're allowed to give more than one enumeration literal the same underlying value. For example, in the United Kingdom, fall is referred to as autumn. You can cater to both cultures as follows:

```
enum Season { Spring, Summer, Fall, Autumn = Fall, Winter }
```

## Choosing an enumeration's underlying type

When you declare an enumeration, the enumeration literals are given values of type `int`. You can also choose to base your enumeration on a different underlying integer type. For example, to declare that the underlying type for `Season` is a `short` rather than an `int`, you can write this:

```
enum Season : short { Spring, Summer, Fall, Winter }
```

The main reason for using `short` is to save memory; an `int` occupies more memory than a `short`, and if you don't need the entire range of values available to an `int`, using a smaller data type can make sense.

You can base an enumeration on any of the eight integer types: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`. The values of all the enumeration literals must fit within the range of the chosen

base type. For example, if you base an enumeration on the byte data type, you can have a maximum of 256 literals (starting at 0).

Now that you know how to declare an enumeration, the next step is to use it. In the following exercise, you'll work with a console application to declare and use an enumeration that represents the months of the year.

### To create and use an enumeration

1. Start Microsoft Visual Studio 2022 if it is not already running.
2. Open the **StructsAndEnums** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 9\StructsAndEnums** folder in your **Documents** folder.

3. In the Code and Text Editor window, display the Month.cs file.

The source file is empty apart from the declaration of a namespace called `StructsAndEnums` and a `// TODO: comment`.

4. Replace the `// TODO: comment` with the enumeration named `Month` within the `StructsAndEnums` namespace, as shown in bold in the code that follows. This enumeration models the months of the year. The 12 enumeration literals for `Month` are January through December.

```
namespace StructsAndEnums
{
 enum Month
 {
 January, February, March, April,
 May, June, July, August,
 September, October, November, December
 }
}
```

5. Display the Program.cs file in the Code and Text Editor window.

As in the exercises in previous chapters, the `Main` method calls the `doWork` method and traps any exceptions that occur.

6. In the Code and Text Editor window, add a statement to the `doWork` method to declare a variable named `first` of type `Month` and initialize it to `Month.January`. Add another statement to write the value of the `first` variable to the console.

The `doWork` method should look like this:

```
static void doWork()
{
 Month first = Month.January;
 Console.WriteLine(first);
}
```



**Note** When you type the period following `Month`, Microsoft IntelliSense automatically displays all the values in the `Month` enumeration.

7. On the **Debug** menu, select **Start Without Debugging**.

Visual Studio 2022 builds and runs the program. Confirm that the word `January` is written to the console.

8. Press **Enter** to close the program and return to the Visual Studio 2022 programming environment.
9. Add two more statements to the `doWork` method to increment the first variable and display its new value to the console, as shown in bold here:

```
static void doWork()
{
 Month first = Month.January;
 Console.WriteLine(first);
 first++;
 Console.WriteLine(first);
}
```

10. On the **Debug** menu, select **Start Without Debugging**.

Visual Studio 2022 builds and runs the program. Confirm that the words `January` and `February` are written to the console.

Notice that performing a mathematical operation (such as the increment operation) on an enumeration variable changes the internal integer value of the variable. When the variable is written to the console, the corresponding enumeration value is displayed.

11. Press **Enter** to close the program and return to the Visual Studio 2022 programming environment.
12. Modify the first statement in the `doWork` method to initialize the `first` variable to `Month.December`, as shown in bold here:

```
static void doWork()
{
 Month first = Month.December;
 Console.WriteLine(first);
 first++;
 Console.WriteLine(first);
}
```

13. On the **Debug** menu, select **Start Without Debugging**.

Visual Studio 2022 builds and runs the program. This time, the word `December` is written to the console, followed by the number 12.



Although you can perform arithmetic on an enumeration, if the results of the operation are outside the range of values defined for the enumeration, all the runtime can do is interpret the value of the variable as the corresponding integer value.

14. Press **Enter** to close the program and return to the Visual Studio 2022 programming environment.

## Working with structures

---

Chapter 8 illustrated that classes define reference types that are always created on the heap. In some cases, the class can contain so little data that the overhead of managing the heap becomes disproportionate. In these cases, it's better to define the type as a structure. A structure is a value type. Because structures are stored on the stack, as long as the structure is reasonably small, the memory-management overhead is often reduced. Like a class, a structure can have its own fields, methods, and (with one important exception discussed later in this chapter) constructors.

### Common structure types

You might not have realized it, but you have already used structures in previous exercises in this book. For example, tuples are actually examples of the `System.ValueTuple` structure. Rather more interestingly, in C#, the primitive numeric types `int`, `long`, and `float` are aliases for the structures `System.Int32`, `System.Int64`, and `System.Single`, respectively. These structures have fields and methods, and you can actually call methods on variables and literals of these types. For example, all these structures provide a `ToString` method that can convert a numeric value to its string representation. The following statements are all legal in C#:

```
int i = 55;
Console.WriteLine(i.ToString());
Console.WriteLine(55.ToString());

float f = 98.765F;
Console.WriteLine(f.ToString());
Console.WriteLine(98.765F.ToString());

Console.WriteLine((500, 600).ToString()); // (500, 600) is a constant tuple
```

You don't see this use of the `ToString` method often because the `Console.WriteLine` method calls it automatically when it's needed. It's more common to use some of the static methods exposed by these structures. For example, in earlier chapters, you used the static `int.Parse` method to convert a string to its corresponding integer value. What you are actually doing is invoking the `Parse` method of the `Int32` structure:

```
string s = "42";
int i = int.Parse(s); // exactly the same as Int32.Parse
```

These structures also include some useful static fields. For example, `Int32.MaxValue` is the maximum value that an `int` can hold, and `Int32.MinValue` is the minimum value that you can store in an `int`.

The following table shows the primitive types in C# and their equivalent types in .NET. Notice that the string and object types are classes (reference types) rather than structures.

Keyword	Type equivalent	Class or structure
<code>bool</code>	<code>System.Boolean</code>	Structure
<code>byte</code>	<code>System.Byte</code>	Structure
<code>decimal</code>	<code>System.Decimal</code>	Structure
<code>double</code>	<code>System.Double</code>	Structure
<code>float</code>	<code>System.Single</code>	Structure
<code>int</code>	<code>System.Int32</code>	Structure
<code>long</code>	<code>System.Int64</code>	Structure
<code>object</code>	<code>System.Object</code>	Class
<code>sbyte</code>	<code>System.SByte</code>	Structure
<code>short</code>	<code>System.Int16</code>	Structure
<code>string</code>	<code>System.String</code>	Class
<code>uint</code>	<code>System.UInt32</code>	Structure
<code>ulong</code>	<code>System.UInt64</code>	Structure
<code>ushort</code>	<code>System.UInt16</code>	Structure

## Declaring a structure

To declare your own structure type, you use the `struct` keyword followed by the name of the type and then enclose the body of the structure between opening and closing braces. Syntactically, the process is similar to declaring a class. For example, here is a structure named `Time` that contains three public `int` fields named `hours`, `minutes`, and `seconds`:

```
struct Time
{
 public int hours, minutes, seconds;
}
```

As with classes, making the fields of a structure public is not advisable in most cases; there's no way to control the values held in public fields. For example, anyone could set the value of `minutes` or

seconds to a value greater than 60. A better idea is to make the fields private and provide your structure with constructors and methods to initialize and manipulate these fields, as shown in this example:

```
struct Time
{
 private int hours, minutes, seconds;
 ...
 public Time(int hh, int mm, int ss)
 {
 this.hours = hh % 24;
 this.minutes = mm % 60;
 this.seconds = ss % 60;
 }

 public int Hours()
 {
 return this.hours;
 }
}
```



**Note** By default, you cannot use many of the common operators on your own structure types. For example, you cannot use the equality operator (==) or the inequality operator (!=) to compare structure type variables. However, you can use the built-in `Equals()` method exposed by all structures to compare structure type variables. You can also explicitly declare and implement operators for your own structure types. The syntax for doing this is covered in Chapter 21, “Querying in-memory data by using query expressions.”

When you copy a value type variable, you get two copies of the value. In contrast, when you copy a reference type variable, you get two references to the same object. In summary, use structures for small data values for which it’s just as or nearly as efficient to copy the value as it would be to copy an address. Use classes for more complex data that is too big to copy efficiently.



**Tip** Use structures to implement simple concepts whose main feature is their value rather than the functionality they provide.

## Understanding differences between structures and classes

A structure and a class are syntactically similar, but they have a few important differences. Let’s look at some of these variations:

- You can’t declare a default constructor (a constructor with no parameters) for a structure. The following example would compile if `Time` were a class, but because `Time` is a structure, it does not:

```
struct Time
{
 public Time() { ... } // compile-time error
 ...
}
```

You can't declare your own default constructor for a structure because the compiler *always* generates one. In a class, the compiler synthesizes the default constructor only if you don't write a constructor yourself. The compiler-generated default constructor for a structure always sets the fields to 0, false, or null—just as for a class. Therefore, you should ensure that a structure value created by the default constructor behaves logically and makes sense with these default values. This has some ramifications that you'll explore in the next exercise.

You can initialize fields to different values by providing a nondefault constructor. However, when you do this, your nondefault constructor must explicitly initialize all fields in your structure; the default initialization no longer occurs. If you fail to do this, you'll get a compile-time error. For example, although the following example would compile and silently initialize seconds to 0 if Time were a class, it fails to compile because Time is a structure:

```
struct Time
{
 private int hours, minutes, seconds;
 ...
 public Time(int hh, int mm)
 {
 this.hours = hh;
 this.minutes = mm;
 } // compile-time error: seconds not initialized
}
```

- In a class, you can initialize instance fields at their point of declaration; in a structure, you cannot. The following example would compile if Time were a class, but it causes a compile-time error because Time is a structure:

```
struct Time
{
 private int hours = 0; // compile-time error
 private int minutes; private int seconds;
 ...
}
```

The following table summarizes the main differences between a structure and a class:

Question	Structure	Class
Is this a value type or a reference type?	A structure is a value type.	A class is a reference type.
Do instances live on the stack or the heap?	Structure instances are called <i>values</i> and usually live on the stack (see the following note).	Class instances are called <i>objects</i> and usually live on the heap.
Can you declare a default constructor?	No	Yes
If you declare your own constructor, will the compiler still generate the default constructor?	Yes	No
If you don't initialize a field in your own constructor, will the compiler automatically initialize it for you?	No	Yes
Are you allowed to initialize instance fields at their point of declaration?	No	Yes



**Note** There are some occasions when a struct will be created on the heap rather than the stack. This most commonly happens when a class declares a field that is a struct type. When an object of such a class is created, the memory for the struct is allocated as part of the object on the heap rather than separately on the stack.

There are other differences between classes and structures concerning inheritance. These differences are covered in Chapter 12, “Working with inheritance.”

## Declaring structure variables

After you’ve defined a structure type, you can use it in the same way you do any other type. For example, if you have defined the `Time` structure, you can create variables, fields, and parameters of type `Time`, as shown in this example:

```
struct Time
{
 private int hours, minutes, seconds;
 ...
}

class Example
{
 private Time currentTime;

 public void Method(Time parameter)
 {
 Time localVariable;
 ...
 }
}
```



**Note** As with enumerations, you can create a nullable version of a structure variable by using the `?` modifier. You can then assign the `null` value to the variable, like so:

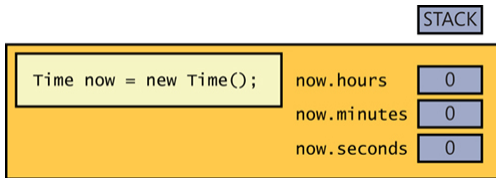
```
Time? currentTime = null;
```

## Understanding structure initialization

Earlier in this chapter, you saw how you could initialize the fields in a structure by using a constructor. If you call a constructor, the various rules described earlier guarantee that all the fields in the structure will be initialized:

```
var now = new Time();
```

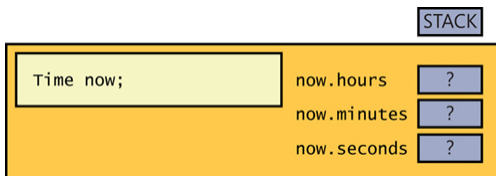
The following illustration depicts the state of the fields in this structure:



However, because structures are value types, you can also create structure variables without calling a constructor, as shown in the following example:

```
Time now;
```

This time, the variable is created, but its fields are left in their uninitialized state. The following illustration depicts the state of the fields in the `now` variable. Any attempt to read the structure or access the values in these fields will result in the compiler error "Use of unassigned local variable."



**Note** In both cases, the `now` variable is created on the stack.

If you've written your own structure constructor, you can also use that to initialize a structure variable. As explained earlier in this chapter, a structure constructor must always explicitly initialize all its fields. For example:

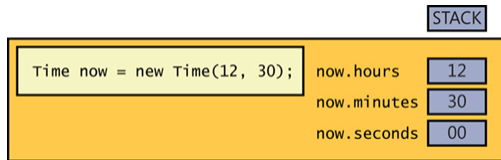
```
struct Time
{
 private int hours, minutes, seconds;
 ...

 public Time(int hh, int mm)
 {
 hours = hh;
 minutes = mm;
 seconds = 0;
 }
}
```

The following example initializes `now` by calling a user-defined constructor:

```
Time now = new Time(12, 30);
```

The following illustration shows the effect of this example:



It's time to put this knowledge into practice. In the following exercise, you'll create and use a structure to represent a date.

### To create and use a structure type

1. In the `StructsAndEnums` project, display the `Date.cs` file in the Code and Text Editor window.
2. Replace the `// TODO: comment` with a structure named `Date` inside the `StructsAndEnums` namespace.

This structure should contain three private fields: one named `year` of type `int`, one named `month` of type `Month` (using the enumeration you created in the preceding exercise), and one named `day` of type `int`. The `Date` structure should look exactly as follows:

```
struct Date
{
 private int year;
 private Month month;
 private int day;
}
```

Consider the default constructor that the compiler will generate for `Date`. This constructor sets the year to 0, the month to 0 (the value of January), and the day to 0. The year value 0 is not valid (because there was no year 0), and the day value 0 is also not valid (because each month starts on day 1). One way to fix this problem is to translate the year and day values by implementing the `Date` structure so that when the year field holds the value  $Y$ , this value represents the year  $Y + 1900$  (you can pick a different century if you prefer, but most significant events in my life occurred in the last century), and when the day field holds the value  $D$ , this value represents the day  $D + 1$ . The default constructor will then set the three fields to values that represent the date 1 January 1900.

If you could override the default constructor and write your own, this would not be an issue because you could then initialize the year and day fields directly to valid values. You can't do this, though, so you have to implement the logic in your structure to translate the compiler-generated default values into meaningful values for your problem domain.

Although you cannot override the default constructor, it's still good practice to define non-default constructors to allow a user to explicitly initialize the fields in a structure to meaningful nondefault values.

3. Add a public constructor to the `Date` structure. This constructor should take three parameters: an `int` named `ccyy` for the year, a `Month` named `mm` for the month, and an `int` named `dd` for the day. Use these three parameters to initialize the corresponding fields. A year field with the value `Y` represents the year `Y + 1900`, so you must initialize the year field to the value `ccyy - 1900`. A day field with the value `D` represents the day `D + 1`, so you need to initialize the day field to the value `dd - 1`.

The `Date` structure should now look like this (with the constructor shown in bold):

```
struct Date
{
 private int year;
 private Month month;
 private int day;

 public Date(int ccyy, Month mm, int dd)
 {
 this.year = ccyy - 1900;
 this.month = mm;
 this.day = dd - 1;
 }
}
```

4. Add a public method named `ToString` to the `Date` structure after the constructor. This method takes no arguments and returns a string representation of the date. Remember, the value of the year field represents year + 1900, and the value of the day field represents day + 1.



**Note** The `ToString` method is a little different from the methods you've seen so far. Every type, including structures and classes that you define, automatically has a `ToString` method whether you want it or not. Its default behavior is to convert the data in a variable to a string representation of that data. Sometimes the default behavior is meaningful; other times it is less so. For example, the default behavior of the `ToString` method generated for the `Date` structure simply generates the string `StructsAndEnums.Date`. To quote Zaphod Beeblebrox in *The Restaurant at the End of the Universe* by Douglas Adams (Pan Macmillan, 1980), this is "shrewd, but dull." You need to define a new version of this method that overrides the default behavior by using the `override` keyword. The topic of overriding methods is discussed in more detail in Chapter 12.

The `ToString` method should look like this:

```
struct Date
{
 ...
 public override string ToString()
 {
 string data = $"{this.month} {this.day + 1} {this.year + 1900}";
 return data;
 }
}
```



In this method, you build a formatted string using the text representations of the values of the `month` field, the expression `this.day + 1`, and the expression `this.year + 1900`. The `ToString` method returns the formatted string as its result.

5. Display the `Program.cs` file in the Code and Text Editor window.
6. In the `doWork` method, comment out the four existing statements.
7. Add statements to the `doWork` method that declare a local variable named `defaultDate` and initialize it to a `Date` value constructed by using the default `Date` constructor. Add another statement to `doWork` to display the `defaultDate` variable on the console by calling `Console.WriteLine`.



**Note** The `Console.WriteLine` method automatically calls the `ToString` method of its argument to format the argument as a string.

The `doWork` method should now look like this:

```
static void doWork()
{
 ...
 var defaultDate = new Date();
 Console.WriteLine(defaultDate);
}
```



**Note** When you type `new Date()`, IntelliSense automatically detects that two constructors are available for the `Date` type.

8. On the **Debug** menu, select **Start Without Debugging** to build and run the program. Verify that the date January 1 1900 is written to the console.
9. Press the **Enter** key to return to the Visual Studio 2022 programming environment.
10. In the Code and Text Editor window, return to the `doWork` method and add two more statements. In the first statement, declare a local variable named `weddingAnniversary` and initialize it to July 4, 2021. (I actually did get married on American Independence Day, although it was many years ago and in the UK.) In the second statement, write the value of `weddingAnniversary` to the console.

The `doWork` method should now look like this:

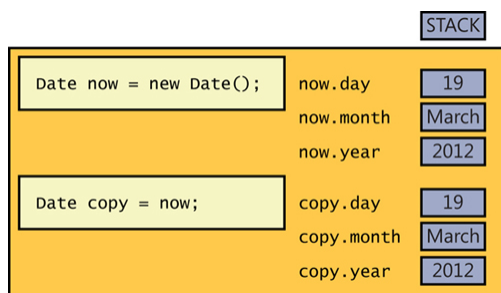
```
static void doWork()
{
 ...
 Date weddingAnniversary = new Date(2021, Month.July, 4);
 Console.WriteLine(weddingAnniversary);
}
```

11. On the **Debug** menu, select **Start Without Debugging**, and then confirm that the date July 4 2021 is written to the console below the previous information.
12. Press **Enter** to close the program and return to Visual Studio 2022.

## Copying structure variables

You're allowed to initialize or assign one structure variable to another structure variable, but only if the structure variable on the right side is completely initialized (that is, if all its fields are populated with valid data rather than undefined values). The following example compiles because `now` is fully initialized. The illustration shows the results of performing such an assignment.

```
Date now = new Date(2012, Month.March, 19);
Date copy = now;
```



The following example fails to compile because `now` is not initialized:

```
Date now;
Date copy = now; // compile-time error: now has not been assigned
```

When you copy a structure variable, each field on the left side is initialized directly using the corresponding field on the right side. This copying is done as a single fast operation that copies the contents of the entire structure, and it never throws an exception. Compare this behavior with the equivalent action if `Time` were a class, in which case both variables (`now` and `copy`) would end up referencing the same object on the heap.



**Note** If you are a C++ programmer, note that this copy behavior cannot be customized.

In the next exercise, you'll contrast the copy behavior of a structure with that of a class.

## To compare the behavior of a structure and a class

1. In the StructsAndEnums project, display the Date.cs file in the Code and Text Editor window.
2. Add the following method to the Date structure. This method moves the date in the structure forward by one month. If, after advancing the month, the value of the month field has moved beyond December, the code resets the month to January and advances the value of the year field by 1.

```
struct Date
{
 ...
 public void AdvanceMonth()
 {
 this.month++;
 if (this.month == Month.December + 1)
 {
 this.month = Month.January;
 this.year++;
 }
 }
}
```

3. Display the Program.cs file in the Code and Text Editor window.
4. In the doWork method, comment out the first two uncommented statements that create and display the value of the defaultDate variable.
5. Add the following code shown in bold to the end of the doWork method. This code creates a copy of the weddingAnniversary variable called weddingAnniversaryCopy and prints out the value of this new variable.

```
static void doWork()
{
 ...
 Date weddingAnniversaryCopy = weddingAnniversary;
 Console.WriteLine($"Value of copy is {weddingAnniversaryCopy}");
}
```

6. Add the following statements shown in bold to the end of the doWork method. These statements call the AdvanceMonth method of the weddingAnniversary variable and then display the value of the weddingAnniversary and weddingAnniversaryCopy variables:

```
static void doWork()
{
 ...
 weddingAnniversary.AdvanceMonth();
 Console.WriteLine($"New value of weddingAnniversary is {weddingAnniversary}");
 Console.WriteLine($"Value of copy is still {weddingAnniversaryCopy}");
}
```

7. On the **Debug** menu, select **Start Without Debugging** to build and run the application. Verify that the console window displays the following messages:

```
July 4 2021
Value of copy is July 4 2021
New value of weddingAnniversary is August 4 2021
Value of copy is still July 4 2021
```

The first message displays the initial value of the `weddingAnniversary` variable (July 4 2021). The second message displays the value of the `weddingAnniversaryCopy` variable. You can see that it contains the same date held in the `weddingAnniversary` variable (July 4 2021). The third message displays the value of the `weddingAnniversary` variable after changing the month to August (August 4 2021). The final statement displays the value of the `weddingAnniversaryCopy` variable. Notice that it has not changed from its original value of July 4 2021.

If `Date` were a class, creating a copy would reference the same object in memory as the original instance. Changing the month in the original instance would therefore also change the date referenced through the copy. You'll verify this assertion in the following steps.

8. Press **Enter** and return to Visual Studio 2022.
9. Display the `Date.cs` file in the Code and Text Editor window.
10. Change the `Date` structure to a class, as shown in bold in the following code example:

```
class Date
{
 ...
}
```

11. On the **Debug** menu, select **Start Without Debugging** to build and run the application again. Verify that the console window displays the following messages:

```
July 4 2021
Value of copy is July 4 2021
New value of weddingAnniversary is August 4 2021
Value of copy is still August 4 2021
```

The first three messages are the same as before. However, the fourth message shows that the value of the `weddingAnniversaryCopy` variable has changed to August 4 2021.

12. Press **Enter** and return to Visual Studio 2022.

## Handling large structures

While most structures will be small in size, you can create structures that contain many fields and/or other `struct` types. When you pass a variable of this type as a parameter to a method, the entire structure, including any child structures, will be copied to the stack. This can affect the performance of your applications if they do this often.

To counter this, pass the structure by reference using the `ref` modifier, as described in Chapter 8. However, be aware that changes made to the values held by fields in the structure will now be permanent; they are not made to a local copy. The following code shows an example:

```
struct BigStruct
{
 // Many, many fields here!
 ...
}
...
var myData = new BigStruct();
ProcessData(ref myData);
...
void ProcessData(ref BigStruct data)
{
 // data is a reference to myData, not a copy
 ...
}
```

## Summary

In this chapter, you saw how to create and use enumerations and structures. You learned some of the similarities and differences between a structure and a class, and you saw how to define constructors to initialize the fields in a structure. You also saw how to represent a structure as a string by overriding the `ToString` method.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 10, “Using arrays.”
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

# Quick reference

To	Do this
Declare an enumeration	Write the keyword <code>enum</code> , followed by the name of the type, followed by a pair of braces containing a comma-separated list of the enumeration literal names. For example: <code>enum Season { Spring, Summer, Fall, Winter }</code>
Declare an enumeration variable	Write the name of the enumeration on the left followed by the name of the variable, followed by a semicolon. For example: <code>Season currentSeason;</code>
Assign an enumeration variable to a value	Write the name of the enumeration literal in combination with the name of the enumeration to which it belongs. For example: <code>currentSeason = Season.Spring; // correct</code> But note that: <code>currentSeason = Spring; // error</code>
Declare a structure type	Write the keyword <code>struct</code> , followed by the name of the structure type, followed by the body of the structure (the constructors, methods, and fields). For example: <pre>struct Time {     public Time(int hh, int mm, int ss)     { ... }     ...     private int hours, minutes, seconds; }</pre>
Declare a structure variable	Write the name of the structure type, followed by the name of the variable, followed by a semicolon. For example: <code>Time now;</code>
Initialize a structure variable to a value	Initialize the variable to a structure value created by calling the structure constructor. For example: <code>Time lunch = new Time(12, 30, 0);</code>



# Using arrays

**After completing this chapter, you will be able to:**

- Declare array variables.
- Create an array instance.
- Populate an array with a set of data items and access those items.
- Copy arrays.
- Create and use multidimensional arrays.

You have already seen how to create and use variables of many different types. However, all the examples of variables you've seen so far have one thing in common: They hold information about a single item (an `int`, a `float`, a `Circle`, a `Date`, and so on). But what if you need to manipulate a set of items?

One solution is to create a variable for each item in the set, but this leads to some further questions:

- How many variables do you need?
- How should you name them?
- If you need to perform the same operation on each item in the set (such as increment each variable in a set of integers), how would you avoid very repetitive code?

Moreover, using a variable for separate items assumes that you know how many items you will need when you write the program. But how often is this the case? For example, if you're writing an application that reads and processes records from a database, how many records are in the database, and how likely is this number to change?

Arrays provide a mechanism that helps to solve these problems. An *array* is an unordered sequence of items. Unlike the fields in a structure or class, which can have different types, all the items in an array have the same type. Also, unlike fields in a structure or class, which are accessed by name, the items in an array live in a contiguous block of memory and are accessed by using an index.



## Declaring array variables

You declare an array variable by specifying the name of the element type, followed by a pair of square brackets, followed by the variable name. The square brackets signify that the variable is an array. For example, to declare an array of `int` variables named `pins` (for holding a set of personal identification numbers), you can write the following:

```
int[] pins; // Personal Identification Numbers
```



**Note** If you're a Microsoft Visual Basic programmer, notice that the declaration uses square brackets, not parentheses. If you're familiar with C and C++, also notice that the size of the array is not part of the declaration. Additionally, the square brackets must be placed before the variable name.

You are not restricted to using primitive types as array elements. You can also create arrays of structures, enumerations, and classes. For example, you can create an array of `Date` structures like this:

```
Date[] dates;
```



**Tip** It's often useful to give array variables plural names, such as `places` (where each element is a `Place`), `people` (where each element is a `Person`), or `times` (where each element is a `Time`).

## Creating an array instance

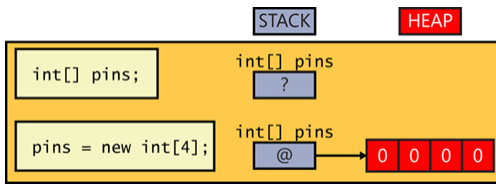
Arrays are reference types, regardless of the type of their elements. This means that an array variable *refers* to a contiguous block of memory holding the array elements on the heap, just as a class variable refers to an object on the heap. (For a description of values and references and the differences between the stack and the heap, see Chapter 8, "Understanding values and references.") This rule applies regardless of the type of the data items in the array. Even if the array contains a value type such as `int`, the memory will still be allocated on the heap; this is the one case where value types are not allocated memory on the stack.

Remember that when you declare a class variable, memory is not allocated for the object until you create the instance by using the `new` keyword. Arrays follow the same pattern: When you declare an array variable, you don't declare its size, and no memory is allocated (other than to hold the reference on the stack). The array is given memory only when the instance is created; this is also the point at which you specify the size of the array.

To create an array instance, you use the `new` keyword followed by the element type, followed by the size of the array you're creating enclosed between square brackets. Creating an array also initializes its elements by using the now-familiar default values (`0`, `null`, or `false`, depending on whether the type is numeric, a reference, or a Boolean, respectively). For example, to create and initialize a new array of four integers for the `pins` variable declared earlier, you write this:

```
pins = new int[4];
```

The following illustration shows what happens when you declare an array, and later when you create an instance of the array:



Because the memory for the array instance is allocated dynamically, the size of the array does not have to be a constant; it can be calculated at runtime, as shown in this example:

```
int size = int.Parse(Console.ReadLine());
int[] pins = new int[size];
```

You can also create an array whose size is 0. This might sound bizarre, but it's useful for situations in which the size of the array is determined dynamically and could even be 0. An array of size 0 is not a `null` array; it's an array containing zero elements.

## Populating and using an array

When you create an array instance, all the elements of the array are initialized to a default value depending on their type. For example, all numeric values default to 0, objects are initialized to `null`, `DateTime` values are set to the date and time "01/01/0001 00:00:00", and strings are initialized to `null`.

You can modify this behavior and initialize the elements of an array to specific values if you prefer. You do this by providing a comma-separated list of values between a pair of braces. For example, to initialize `pins` to an array of four `int` variables whose values are 9, 3, 7, and 2, you write this:

```
int[] pins = new int[4]{ 9, 3, 7, 2 };
```

The values between the braces don't have to be constants; they can be values calculated at runtime, as shown in the following example, which populates the `pins` array with four random numbers:

```
Random r = new Random();
int[] pins = new int[4]{ r.Next() % 10, r.Next() % 10, r.Next() % 10, r.Next() % 10 };
```



**Note** The `System.Random` class is a pseudorandom number generator. The `Next` method returns a nonnegative random integer in the range 0 to `Int32.MaxValue` by default. The `Next` method is overloaded, and other versions enable you to specify the minimum value and maximum value of the range. The default constructor for the `Random` class seeds the random number generator with a time-dependent seed value, which reduces the possibility of the class duplicating a sequence of random numbers. Using an overloaded version of the constructor, you can provide your own seed value. That way, you can generate a repeatable sequence of random numbers for testing purposes.

The number of values between the braces must exactly match the size of the array instance being created:

```
int[] pins = new int[3]{ 9, 3, 7, 2 }; // compile-time error
int[] pins = new int[4]{ 9, 3, 7 }; // compile-time error
int[] pins = new int[4]{ 9, 3, 7, 2 }; // OK
```

When you're initializing an array variable in this way, you can actually omit the `new` expression and the size of the array. In this case, the compiler calculates the size from the number of initializers and generates code to create the array, such as in the following example:

```
int[] pins = { 9, 3, 7, 2 };
```

If you create an array of structures or objects, you can initialize each structure in the array by calling the structure or class constructor, as shown in this example:

```
Time[] schedule = { new Time(12,30), new Time(5,30) };
```

## Creating an implicitly typed array

The element type when you declare an array must match the type of elements that you attempt to store in the array. For example, if you declare `pins` to be an array of `int`, as shown in the preceding examples, you cannot store a `double`, `string`, `struct`, or anything that is not an `int` in this array.

If you specify a list of initializers when declaring an array, you can let the C# compiler infer the actual type of the elements in the array for you, like this:

```
var names = new[]{"John", "Diana", "James", "Francesca"};
```

In this example, the C# compiler determines that the `names` variable is an array of strings. It's worth pointing out a couple of syntactic quirks in this declaration:

- You omit the square brackets from the type; the `names` variable in this example is declared simply as `var`, not `var []`.
- You must specify the `new` operator and square brackets before the initializer list.

If you use this syntax, you must ensure that all the initializers have the same type. This next example causes the compile-time error "No best type found for implicitly-typed array":

```
var bad = new[]{"John", "Diana", 99, 100};
```

However, in some cases, the compiler will convert elements to a different type, if doing so makes sense. In the following code, the `numbers` array is an array of `double` because the constants `3.5` and `99.999` are both `double`, and the C# compiler can convert the integer values `1` and `2` to `double` values:

```
var numbers = new[]{1, 2, 3.5, 99.999};
```



**Note** Generally, it's best to avoid mixing types, hoping that the compiler will convert them for you.

Implicitly typed arrays are most useful when you're working with anonymous types, as described in Chapter 7, "Creating and managing classes and objects." The following code creates an array of anonymous objects, each containing two fields specifying the name and age of the members of my family. (The fields in the anonymous types must be the same for each element of the array.)

```
var names = new[] { new { Name = "John", Age = 57 },
 new { Name = "Diana", Age = 57 },
 new { Name = "James", Age = 30 },
 new { Name = "Francesca", Age = 26 } };
```

## Accessing an individual array element

To access an individual array element, you must provide an index indicating which element you require. Array indexes are zero-based; so, the initial element of an array lives at index 0 and not index 1. An index value of 1 accesses the second element. For example, you can read the contents of element 2 (the third element) of the `pins` array into an `int` variable by using the following code:

```
int myPin;
myPin = pins[2];
```

Similarly, you can change the contents of an array by assigning a value to an indexed element:

```
myPin = 1645;
pins[2] = myPin;
```

All array element access is bounds-checked. If you specify an index that is less than 0 or greater than or equal to the length of the array, the compiler throws an `IndexOutOfRangeException` exception, as in this example:

```
try
{
 int[] pins = { 9, 3, 7, 2 };
 Console.WriteLine(pins[4]); // error, the fourth and last element is at index 3
}
catch (IndexOutOfRangeException ex)
{
 ...
}
```

## Accessing a series of array elements

`C#` lets you retrieve a contiguous set of elements from an array using an index of the form `x..y`. The sequence beginning at element `x` and finishing at `y` but—and this is important—not including `y` are returned. The result is another array containing the elements retrieved. In the following example, the array `subset` contains elements 0 and 1 from the `names` array:

```
var names = new[] { new { Name = "John", Age = 57 },
 new { Name = "Diana", Age = 57 },
 new { Name = "James", Age = 30 },
 new { Name = "Francesca", Age = 26 } };

var subset = names[0..2];
```

The subset array will contain the values for John and Diana (elements 0 and 1). The first number of the range must be less than or equal to the second number; otherwise, you will get a `System.ArgumentOutOfRangeException` exception. If the first number in the range and second number are the same, the result is a zero-length array.

You can also specify an expression of the form `^x` as the starting element. In this case, `y - 1` elements are retrieved counting back from element `x` (the elements are still retrieved in forward order, though). In the following statement, the subset array contains the values for Diana, James, and Francesca (starting backward from element 3, or `4 - 1` elements):

```
var subset = names[^3..4];
```

## Iterating through an array

All arrays are actually instances of the `System.Array` class in the .NET libraries, and this class defines some useful properties and methods. For example, you can query the `Length` property to discover how many elements an array contains and iterate through all the elements of an array by using a `for` statement. The following sample code writes the array element values of the `pins` array to the console:

```
int[] pins = { 9, 3, 7, 2 };
for (int index = 0; index < pins.Length; index++)
{
 int pin = pins[index];
 Console.WriteLine(pin);
}
```



**Note** `Length` is a property and not a method, which is why you don't use parentheses when you call it. You'll learn about properties in Chapter 15, "Implementing properties to access fields."

It's common for new programmers to forget that arrays start at element 0 and that the last element is numbered `Length - 1`. Fortunately, C# provides the `foreach` statement, with which you can iterate through the elements of an array without worrying about these issues. For example, here's the preceding `for` statement rewritten as an equivalent `foreach` statement:

```
int[] pins = { 9, 3, 7, 2 };

foreach (int pin in pins)
{
 Console.WriteLine(pin);
}
```

The `foreach` statement declares an iteration variable (in this example, `int pin`) that automatically acquires the value of each element in the array. The type of this variable must match the type of the elements in the array. The `foreach` statement is the preferred way to iterate through an array;

it expresses the intention of the code directly, and all of the for loop scaffolding drops away. However, in a few cases, you'll find that you must revert to a for statement:

- A foreach statement always iterates through the entire array. If you want to iterate through only a known portion of an array (for example, the first half) or bypass certain elements (for example, every third element), it's easier to use a for statement.
- A foreach statement always iterates from index 0 through index Length - 1. If you want to iterate backward or in some other sequence, it's easier to use a for statement.
- If the body of the loop needs to know the index of the element rather than just the value of the element, you must use a for statement.
- If you need to modify the elements of the array, you must use a for statement. This is because the iteration variable of the foreach statement is a read-only copy of each element of the array.



**Tip** It's perfectly safe to attempt to iterate through a zero-length array by using a foreach statement.

You can declare the iteration variable as a var and let the C# compiler determine the type of the variable from the type of the elements in the array. This is especially useful if you don't actually know the type of the elements in the array, such as when the array contains anonymous objects. The following example demonstrates how you can iterate through the array of family members shown earlier:

```
var names = new[] { new { Name = "John", Age = 57 },
 new { Name = "Diana", Age = 57 },
 new { Name = "James", Age = 30 },
 new { Name = "Francesca", Age = 26 } };

foreach (var familyMember in names)
{
 Console.WriteLine($"Name: {familyMember.Name}, Age: {familyMember.Age}");
}
```

## Passing arrays as parameters or return values for a method

You can define methods that take arrays as parameters or pass them back as return values.

The syntax for passing an array as a parameter is much the same as for declaring an array. For example, the code sample that follows defines a method named ProcessData that takes an array of integers as a parameter. The body of the method iterates through the array and performs some unspecified processing on each element:

```
public void ProcessData(int[] data)
{
 foreach (int i in data)
 {
 ...
 }
}
```

Arrays are reference objects, so if you modify the contents of an array passed as a parameter inside a method such as `ProcessData`, the modification is visible through all references to the array, including the original argument passed as the parameter.

To return an array from a method, you specify the type of the array as the return type. In the method, you create and populate the array. The following example prompts the user for the size of an array, followed by the data for each element. The array created by the method is passed back as the return value:

```
public int[] ReadData()
{
 Console.WriteLine("How many elements?");
 string reply = Console.ReadLine();
 int numElements = int.Parse(reply);

 int[] data = new int[numElements];
 for (int i = 0; i < numElements; i++)
 {
 Console.WriteLine($"Enter data for element {i}");
 reply = Console.ReadLine();
 int elementData = int.Parse(reply);
 data[i] = elementData;
 }
 return data;
}
```

You can call the `ReadData` method like this:

```
int[] data = ReadData();
```

## Array parameters and the `Main` method

You might have noticed that the `Main` method for an application takes an array of strings as a parameter:

```
static void Main(string[] args)
{
 ...
}
```

Remember that the `Main` method is called when your program starts running; it is the entry point of your application. If you start the application from the command line, you can specify additional command-line arguments. The Windows operating system passes these arguments to the common language runtime (CLR), which in turn passes them as arguments to the `Main` method. This mechanism gives you a simple way to allow a user to provide information when an application starts running instead of prompting the user interactively. This approach is useful if you want to build utilities that can be run from automated scripts.

The following example is taken from a utility application called `MyFileUtil` that processes files. It expects a set of file names on the command line and calls the `ProcessFile` method (not shown) to handle each file specified:

```
static void Main(string[] args)
{
 foreach (string filename in args)
 {
 ProcessFile(filename);
 }
}
```

The user can run the `MyFileUtil` application from the command line, like this:

```
MyFileUtil C:\Temp\TestData.dat C:\Users\John\Documents\MyDoc.txt
```

Each command-line argument is separated by a space. It's up to the `MyFileUtil` application to verify that these arguments are valid.

## Copying arrays

Arrays are reference types. (Remember that an array is an instance of the `System.Array` class.) An array variable contains a reference to an array instance. This means that when you copy an array variable, you actually end up with two references to the same array instance, as demonstrated in the following example:

```
int[] pins = { 9, 3, 7, 2 };
int[] alias = pins; // alias and pins refer to the same array instance
```

In this example, if you modify the value at `pins[1]`, the change will also be visible by reading `alias[1]`.

If you want to make a copy of the array instance (the data on the heap) that an array variable refers to, you must do two things:

1. Create a new array instance of the same type and the same length as the array you are copying.
2. Copy the data from the original array, element by element, to the new array, as in this example:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
for (int i = 0; i < pins.Length; i++)
{
 copy[i] = pins[i];
}
```

Note that this code uses the `Length` property of the original array to specify the size of the new array.



Copying an array is actually a common requirement of many applications—so much so that the `System.Array` class provides some useful methods that you can employ to copy an array. For example, the `CopyTo` method copies the contents of one array into another array given a specified starting index. The following example copies all the elements from the `pins` array to the `copy` array starting at element 0:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
pins.CopyTo(copy, 0);
```

Another way to copy the values is to use the `System.Array` static method named `Copy`. As with `CopyTo`, you must initialize the target array before calling `Copy`:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length);
```



**Note** Be sure to specify a valid value for the length parameter of the `Array.Copy` method. If you provide a negative value, the method throws an `ArgumentOutOfRangeException` exception. If you specify a value that is greater than the number of elements in the source array, the method throws an `ArgumentException` exception.

Yet another alternative is to use the `System.Array` instance method named `Clone`. You can call this method to create an entire array and copy it in one action:

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = (int[])pins.Clone();
```



**Note** `Clone` methods were described in Chapter 8. The `Clone` method of the `Array` class returns an object rather than an array, which is why you must cast it to an array of the appropriate type when you use it. Furthermore, the `Clone`, `CopyTo`, and `Copy` methods all create a shallow copy of an array. (Shallow and deep copying are also described in Chapter 8.) If the elements in the array being copied contain references, the `Clone` method simply copies the references rather than the objects being referred to. After copying, both arrays refer to the same set of objects. If you need to create a deep copy of such an array, you must use appropriate code in a `for` loop.

## Using multidimensional arrays

The arrays shown so far have contained a single dimension, and you can think of them as simple lists of values. However, you can create arrays with more than one dimension. For example, to create a two-dimensional array, you specify an array that requires two integer indexes. The following code creates a two-dimensional array of 24 integers called `items`:

```
int[,] items = new int[4, 6];
```



**Tip** If it helps, you can think of the array as a table, with the first dimension specifying a number of rows and the second specifying a number of columns.

To access an element in the array, you provide two index values to specify the “cell” (the intersection of a row and a column) holding the element. The following code shows some examples using the `items` array:

```
items[2, 3] = 99; // set the element at cell(2,3) to 99
items[2, 4] = items [2,3]; // copy the element in cell(2, 3) to cell(2, 4)
items[2, 4]++; // increment the integer value at cell(2, 4)
```

There is no limit on the number of dimensions that you can specify for an array. The next code example creates and uses an array called `cube` that contains three dimensions. Notice that you must specify three indexes to access each element in the array.

```
int[, ,] cube = new int[5, 5, 5];
cube[1, 2, 1] = 101;
cube[1, 2, 2] = cube[1, 2, 1] * 3;
```

At this point, it’s worth offering a word of caution about creating arrays with more than three dimensions. Specifically, arrays can consume a lot of memory. The `cube` array contains 125 elements ( $5 * 5 * 5$ ). A four-dimensional array for which each dimension has a size of 5 contains 625 elements. If you start to create arrays with three or more dimensions, you can soon run out of memory. Therefore, you should always be prepared to catch and handle `OutOfMemoryException` exceptions when you use multidimensional arrays.

## Creating jagged arrays

In C#, ordinary multidimensional arrays are sometimes referred to as *rectangular* arrays. Each dimension has a regular shape. For example, in the following tabular, two-dimensional `items` array, every row has a column containing 40 elements, and there are 160 elements in total:

```
int[,] items = new int[4, 40];
```

As mentioned, multidimensional arrays can consume a lot of memory. If the application uses only some of the data in each column, allocating memory for unused elements is a waste. In this scenario, you can use a *jagged* array, for which each column has a different length, like this:

```
int[][] items = new int[4][];
int[] columnForRow0 = new int[3];
int[] columnForRow1 = new int[10];
int[] columnForRow2 = new int[40];
int[] columnForRow3 = new int[25];
items[0] = columnForRow0;
items[1] = columnForRow1;
items[2] = columnForRow2;
items[3] = columnForRow3;
...
```

In this example, the application requires only three elements in the first column, 10 elements in the second column, 40 elements in the third column, and 25 elements in the final column. This code illustrates an array of arrays; `items`, instead of being a two-dimensional array, has only a single dimension, but the elements in that dimension are themselves arrays. Furthermore, the total size of the `items` array is 78 elements rather than 160; no space is allocated for elements that the application will not use.

It's worth highlighting some of the syntax in this example. The following declaration specifies that `items` is an array of arrays of `int`:

```
int[][] items;
```

The following statement initializes `items` to hold four elements, each of which is an array of indeterminate length:

```
items = new int[4][];
```

The arrays `columnForRow0` to `columnForRow3` are all single-dimensional `int` arrays, initialized to hold the required amount of data for each column.

Finally, each column array is assigned to the appropriate elements in the `items` array, like this:

```
items[0] = columnForRow0;
```

Recall that arrays are reference objects, so this statement simply adds a reference to `columnForRow0` to the first element in the `items` array; it does not actually copy any data. You can populate data in this column either by assigning a value to an indexed element in `columnForRow0` or by referencing it through the `items` array. The following statements are equivalent:

```
columnForRow0[1] = 99;
items[0][1] = 99;
```

You can extend this idea further if you want to create arrays of arrays of arrays rather than rectangular three-dimensional arrays, and so on.



**Note** If you have written code using Java, you should be familiar with this concept. Java does not have multidimensional arrays; instead, you can create arrays of arrays exactly as just described.

In the following exercise, you'll use arrays to implement an application that deals playing cards as part of a card game. The application displays a form with four hands of cards dealt at random from a regular (52 cards) pack of playing cards. You will complete the code that deals the cards for each hand.

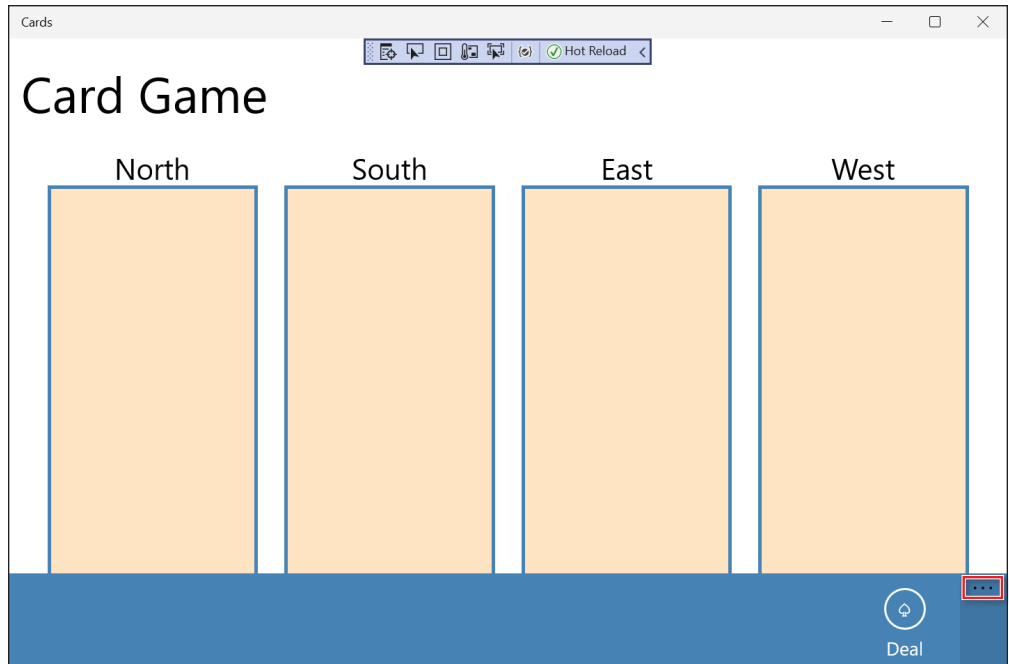
### To use arrays to implement a card game

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **Cards** solution, which is located in the `\Microsoft Press\VCSBS\Chapter 10\Cards` folder in your **Documents** folder.

3. On the **Debug** menu, select **Start Debugging** to build and run the application.

A form appears with the caption Card Game and four text boxes labeled North, South, East, and West. At the bottom is a command bar with an ellipsis (...).

4. Select the ellipsis to expand the command bar. A button with the caption Deal appears.



**Note** The technique used here is the preferred mechanism for locating command buttons in Universal Windows Platform (UWP) apps. From here on, all UWP apps presented in this book will follow this style.

5. Select **Deal**.

Nothing happens. You have not yet implemented the code that deals the cards, which is what you will do in this exercise.

6. Return to Visual Studio 2022. On the **Debug** menu, select **Stop Debugging**.

7. In Solution Explorer, locate the Value.cs file, and open it in the Code and Text Editor window.

This file contains an enumeration called `Value`, which represents the different values that a card can have, in ascending order:

```
enum Value { Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Jack, Queen, King, Ace }
```

8. Open the `Suit.cs` file in the Code and Text Editor window.

This file contains an enumeration called `Suit`, which represents the suits of cards in a regular pack:

```
enum Suit { Clubs, Diamonds, Hearts, Spades }
```

9. Open the `PlayingCard.cs` file in the Code and Text Editor window.

This file contains the `PlayingCard` class. This class models a single playing card.

```
class PlayingCard
{
 private readonly Suit suit;
 private readonly Value value;

 public PlayingCard(Suit s, Value v)
 {
 this.suit = s;
 this.value = v;
 }

 public override string ToString()
 {
 string result = $"{this.value} of {this.suit}";
 return result;
 }

 public Suit CardSuit()
 {
 return this.suit;
 }

 public Value CardValue()
 {
 return this.value;
 }
}
```

This class has two `readonly` fields that represent the value and suit of the card. The constructor initializes these fields.



**Note** A `readonly` field is useful for modeling data that should not change after it has been initialized. You can assign a value to a `readonly` field by using an initializer when you declare it or in a constructor, but thereafter you cannot modify it.

The class contains a pair of methods, `CardValue` and `CardSuit`, that return this information and overrides the `ToString` method to return a string representation of the card.



**Note** The `CardValue` and `CardSuit` methods are actually better implemented as properties, which you'll learn about in Chapter 15.

10. Open the Pack.cs file in the Code and Text Editor window.

This file contains the Pack class, which models a pack of playing cards. At the top of the Pack class are two public const int fields called NumSuits and CardsPerSuit. These two fields specify the number of suits in a pack of cards and the number of cards in each suit.

The private cardPack variable is a two-dimensional array of PlayingCard objects. You'll use the first dimension to specify the suit and the second dimension to specify the value of the card in the suit.

The randomCardSelector variable is a random number generated based on the Random class. You'll use the randomCardSelector variable to help shuffle the cards before they're dealt to each hand.

```
class Pack
{
 public const int NumSuits = 4;
 public const int CardsPerSuit = 13;
 private PlayingCard[,] cardPack;
 private Random randomCardSelector = new Random();
 ...
}
```

11. Locate the default constructor for the Pack class. Currently, this constructor is empty except for a // TODO: comment. Delete the comment and add the following statement shown in bold to instantiate the cardPack array with the appropriate values for each dimension:

```
public Pack()
{
 this.cardPack = new PlayingCard[NumSuits, CardsPerSuit];
}
```

12. Add the following code shown in bold to the Pack constructor. These statements populate the cardPack array with a full, sorted deck of cards.

```
public Pack()
{
 this.cardPack = new PlayingCard[NumSuits, CardsPerSuit];
 for (var suit = Suit.Clubs; suit <= Suit.Spades; suit++)
 {
 for (var value = Value.Two; value <= Value.Ace; value++)
 {
 this.cardPack[(int)suit, (int)value] = new PlayingCard(suit, value);
 }
 }
}
```

The outer for loop iterates through the list of values in the Suit enumeration, and the inner for loop iterates through the values each card can have in each suit. The inner for loop also creates a new PlayingCard object of the specified suit and value and adds it to the appropriate element in the cardPack array.



**Note** You must use one of the integer types as indexes into an array. The `suit` and `value` variables are enumeration variables. However, enumerations are based on the integer types, so it's safe to cast them to `int` as shown in the code.

13. Find the `DealCardFromPack` method in the `Pack` class. The purpose of this method is to pick a random card from the pack, remove the card from the pack to prevent it from being selected again, and then pass it back as the return value from the method.

The first task in this method is to pick a suit at random. Delete the comment and the statement that throws the `NotImplementedException` exception from this method and replace them with the following statement shown in bold:

```
public PlayingCard DealCardFromPack()
{
 var suit = (Suit)randomCardSelector.Next(NumSuits);
}
```

This statement uses the `Next` method of the `randomCardSelector` object to return a random number corresponding to a suit. The parameter to the `Next` method specifies the exclusive upper bound of the range to use; the value selected is between 0 and this value minus 1. Note that the value returned is an `int`, so it must be cast before you can assign it a `Suit` variable.

There's always the possibility that no cards of the selected suit are left. You need to handle this situation and pick another suit if necessary.

14. After the code that selects a suit at random, add the `while` loop that follows (shown in bold):

```
public PlayingCard DealCardFromPack()
{
 var suit = (Suit)randomCardSelector.Next(NumSuits);
 while (this.IsSuitEmpty(suit))
 {
 suit = (Suit)randomCardSelector.Next(NumSuits);
 }
}
```

This loop calls the `IsSuitEmpty` method to determine whether any cards of the specified suit are left in the pack. (You will implement the logic for this method shortly.) If not, it picks another suit at random—or it might actually pick the same suit again—and checks again. The loop repeats the process until it finds a suit with at least one card left.

You have now selected at random a suit with at least one card left. The next task is to pick a card at random in this suit. You can use the random number generator to select a card value, but as before, there's no guarantee that the card with the chosen value has not already been dealt. However, you can use the same idiom as before: Call the `IsCardAlreadyDealt` method (which you'll examine and complete later). You use this method to determine whether the card has already been dealt, and if so, to pick another card at random and try again—repeating the process until a card is found.

15. Add the following statements shown in bold to the `DealCardFromPack` method, after the existing code, to call the `IsCardAlreadyDealt` method:

```
public PlayingCard DealCardFromPack()
{
 ...
 var value = (Value)randomCardSelector.Next(CardsPerSuit);
 while (this.IsCardAlreadyDealt(suit, value))
 {
 value = (Value)randomCardSelector.Next(CardsPerSuit);
 }
}
```

16. You have now selected a random playing card that has not been dealt previously. Add the following code to the end of the `DealCardFromPack` method to return this card and set the corresponding element in the `cardPack` array to `null`:

```
public PlayingCard DealCardFromPack()
{
 ...
 var card = this.cardPack[(int)suit, (int)value];
 this.cardPack[(int)suit, (int)value] = null;
 return card;
}
```

17. Locate the `IsSuitEmpty` method. Remember that the purpose of this method is to take a `Suit` parameter and return a Boolean value indicating whether there are any more cards of this suit left in the pack. Delete the comment and the statement that throws the `NotImplementedException` exception from this method, and then add the following code shown in bold:

```
private bool IsSuitEmpty(Suit suit)
{
 bool result = true;
 for (var value = Value.Two; value <= Value.Ace; value++)
 {
 if (!IsCardAlreadyDealt(suit, value))
 {
 result = false;
 break;
 }
 }

 return result;
}
```

This code iterates through the possible card values and uses the `IsCardAlreadyDealt` method (which you'll complete in the next step) to determine whether there's a card left in the `cardPack` array that has the specified suit and value. If the loop finds a card, the value in the `result` variable is set to `false`, and the `break` statement causes the loop to terminate. If the loop completes without finding a card, the `result` variable remains set to its initial value of `true`. The value of the `result` variable is passed back as the return value of the method.



- 18.** Find the `IsCardAlreadyDealt` method. The purpose of this method is to determine whether the card with the specified suit and value has already been dealt and removed from the pack. You will see later that when the `DealCardFromPack` method deals a card, it removes the card from the `cardPack` array and sets the corresponding element to `null`. Replace the body of this method with the code shown in bold:

```
private bool IsCardAlreadyDealt(Suit suit, Value value)
=> (this.cardPack[(int)suit, (int)value] == null);
```

This method returns `true` if the element in the `cardPack` array corresponding to the suit and value is `null`, and `false` otherwise.

- 19.** The next step is to add the selected playing card to a hand. Open the `Hand.cs` file and display it in the Code and Text Editor window. This file contains the `Hand` class, which implements a hand of cards (that is, all cards dealt to one player).

This file contains a `public const int` field called `HandSize`, which is set to the size of a hand of cards (13). It also contains an array of `PlayingCard` objects, which is initialized by using the `HandSize` constant. Your code will use the `playingCardCount` field to keep track of how many cards the hand currently contains as it is being populated.

```
class Hand
{
 public const int HandSize = 13;
 private PlayingCard[] cards = new PlayingCard[HandSize];
 private int playingCardCount = 0;
 ...
}
```

The `ToString` method generates a string representation of the cards in the hand. It uses a `foreach` loop to iterate through the items in the `cards` array and calls the `ToString` method on each `PlayingCard` object it finds. These strings are concatenated with a newline character in between (using the `Environment.NewLine` constant to specify the newline character) for formatting purposes.

```
public override string ToString()
{
 string result = "";
 foreach (PlayingCard card in this.cards)
 {
 result += $"{card.ToString()}{Environment.NewLine}";
 }

 return result;
}
```

- 20.** Locate the `AddCardToHand` method in the `Hand` class. The purpose of this method is to add the playing card specified as the parameter to the hand. Delete the comment and then add the following statements shown in bold to this method:

```
public void AddCardToHand(PlayingCard cardDealt)
{
 if (this.playingCardCount >= HandSize)
 {
 throw new ArgumentException("Too many cards");
 }
 this.cards[this.playingCardCount] = cardDealt;
 this.playingCardCount++;
}
```

This code checks to ensure that the hand is not already full. If the hand is full, it throws an `ArgumentException` exception. (This should never occur, but it's good practice to check to be safe.) Otherwise, the card is added to the `cards` array at the index specified by the `playingCardCount` variable, and this variable is then incremented.

- 21.** In Solution Explorer, expand the `MainPage.xaml` node and then open the `MainPage.xaml.cs` file in the Code and Text Editor window.

This is the code for the Card Game window.

- 22.** Locate the `dealClick` method. This method runs when the user selects the Deal button. Currently, it contains an empty try block and an exception handler that displays a message if an exception occurs.

- 23.** Delete the comment and then add the following statement shown in bold to the try block:

```
private void dealClick(object sender, RoutedEventArgs e)
{
 try
 {
 pack = new Pack();
 }
 catch (Exception ex)
 {
 ...
 }
}
```

This statement simply creates a new pack of cards. You saw earlier that this class contains a two-dimensional array holding the cards in the pack, and the constructor populates this array with the details of each card. You now need to create four hands of cards from this pack.

- 24.** Add the following statements shown in bold to the try block:

```
try
{
 pack = new Pack();

 for (var handNum = 0; handNum < NumHands; handNum++)
 {
 hands[handNum] = new Hand();
 }
}
catch (Exception ex)
{
 ...
}
```

This for loop creates four hands from the pack of cards and stores them in an array called `hands`. Each hand is initially empty, so you need to deal the cards from the pack to each hand.

- 25.** Add the following code shown in bold to the for loop:

```
try
{
 ...
 for (var handNum = 0; handNum < NumHands; handNum++)
 {
 hands[handNum] = new Hand();
 for (var numCards = 0; numCards < Hand.HandSize; numCards++)
 {
 var cardDealt = pack.DealCardFromPack();
 hands[handNum].AddCardToHand(cardDealt);
 }
 }
}
catch (Exception ex)
{
 ...
}
```

The inner for loop populates each hand by using the `DealCardFromPack` method to retrieve a card at random from the pack and the `AddCardToHand` method to add this card to a hand.

- 26.** Add the following code shown in bold after the outer for loop:

```
try
{
 ...
 for (int handNum = 0; handNum < NumHands; handNum++)
 {
 ...
 }

 north.Text = hands[0].ToString();
 south.Text = hands[1].ToString();
 east.Text = hands[2].ToString();
 west.Text = hands[3].ToString();
}
```

```

}
catch (Exception ex)
{
 ...
}

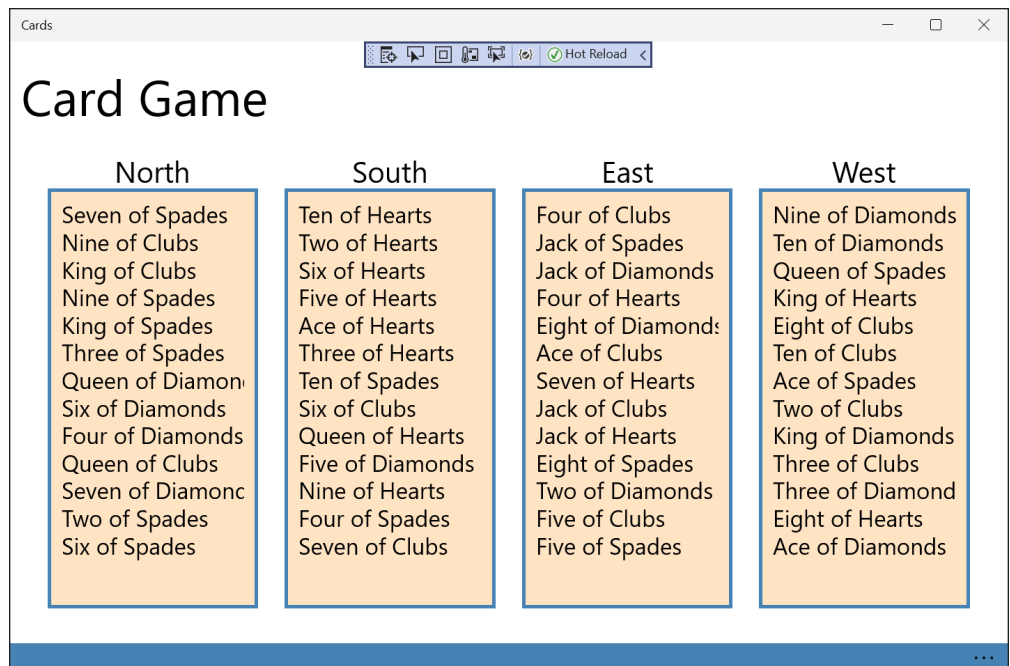
```

When all the cards have been dealt, this code displays each hand in the text boxes on the form. These text boxes are called `north`, `south`, `east`, and `west`. The code uses the `ToString` method of each hand to format the output.

If an exception occurs at any point, the catch handler displays a message box with the error message for the exception.

27. On the **Debug** menu, select **Start Debugging**. When the Card Game window appears, expand the command bar and select **Deal**.

The cards in the pack should be dealt at random to each hand, and the cards in each hand should be displayed on the form, similarly as shown in the following image:



28. Select **Deal** again. Verify that a new set of hands is dealt, and the cards in each hand change.
29. Return to Visual Studio and stop debugging.

## Accessing arrays that contain value types

---

You can think of an array as a simple collection of data, ordered by an index. You can easily retrieve an item if you know its index, but if you want to find data based on some other attribute, then you typically have to implement a helper method that performs the search and returns the index of the required item.

As an example, consider the following code that creates an array of Person objects, where Person is a class:

```
class Person
{
 public string Name;
 public int Age;

 public Person(string name, int age)
 {
 this.Name = name;
 this.Age = age;
 }
}
...
var names = new[] { new { Name = "John", Age = 57 },
 new { Name = "Diana", Age = 57 },
 new { Name = "James", Age = 30 },
 new { Name = "Francesca", Age = 26 } };
```

You want to find the youngest member of the family in the Person array, so you write the following method:

```
Person findYoungest()
{
 int youngest = 0;
 for (int i = 1; i < family.Length; i++)
 {
 if (family[i].Age < family[youngest].Age)
 {
 youngest = i;
 }
 }
 return family[youngest];
}
```

You can then call this method and display the results in the following manner:

```
var mostYouthful = findYoungest();
Console.WriteLine($"Name: {mostYouthful.Name}, Age: {mostYouthful.Age}");
```

Hopefully, this displays the following result:

```
Name: Francesca, Age: 26
```

This is all very satisfactory and works well. Next, you decide that you want to update the age of the youngest family member (Francesca has just had her birthday and is now 27), so you write the following statement:

```
mostYouthful.Age++;
```

Finally, to confirm that everything has been changed correctly, you use the following statements to iterate through the family array and display its contents:

```
foreach (Person familyMember in family)
{
 Console.WriteLine($"Name: {familyMember.Name}, Age: {familyMember.Age}");
}
```

You are pleased to observe that the results are correct, and Francesca's age has been modified:

```
Name: John, Age: 57
Name: Diana, Age: 57
Name: James, Age: 30
Name: Francesca, Age: 27
```

At this point, you rethink the Person class and decide it should really be a struct, so you change it:

```
struct Person
{
 public string Name;
 public int Age;

 public Person(string name, int age)
 {
 this.Name = name;
 this.Age = age;
 }
}
```

Your code compiles and runs, but you notice that Francesca's age is no longer being updated; the output of the foreach loop looks like this:

```
Name: John, Age: 57
Name: Diana, Age: 57
Name: James, Age: 30
Name: Francesca, Age: 27
```

The issue is that you have converted a reference type into a value type. The data in the family array has changed from being a set of references to objects on the heap to copies of data on the stack. The value returned by the findYoungest method was originally a reference to a Person object, and the increment operation on the Age field made through that reference updated the original object on the heap. Now the family array contains value types, and the value returned by the findYoungest method is a copy of the item in the array rather than a reference. So, when the increment operation is performed on the Age field, this operation updates a copy of the Person and not the item stored in the family array.

To handle this situation, you can amend the `findYoungest` method to explicitly return a reference to the value type rather than to a copy. You can achieve this by using the `ref` keyword, as follows:

```
ref Person findYoungest()
{
 int youngest = 0;
 for (int i = 1; i < family.Length; i++)
 {
 if (family[i].Age < family[youngest].Age)
 {
 youngest = i;
 }
 }
 return ref family[youngest];
}
```

Note that most of the code is unchanged. The return type of the method has changed to `ref Person` (a reference to a `Person`), and the return statement similarly states that it passes back a reference to the youngest item in the `family` array.

When you call the method, you must make a couple of corresponding changes:

```
ref var mostYouthful = ref findYoungest();
```

These modifications indicate that `mostYouthful` is a reference to an item in the `family` array. You access the fields in this item in the same way as before; the C# compiler knows that it should reference that data through the variable. The result is that the following increment statement updates the data in the array rather than in a copy:

```
mostYouthful.Age++;
```

When you print the contents of the array, Francesca's age has changed:

```
foreach (Person familyMember in family)
{
 Console.WriteLine($"Name: {familyMember.Name}, Age: {familyMember.Age}");
}
```

```
Name: John, Age: 57
Name: Diana, Age: 57
Name: James, Age: 30
Name: Francesca, Age: 27
```

Returning reference data from a method in this way is a powerful technique, but you must treat it with care. You can only return a reference to data that still exists when the method has finished, such as an element in an array. For example, you cannot return a reference to a local variable created on the stack by the method:

```
// Don't try this; it won't compile
ref int danglingReference()
{
 int i;
 ... // Calculate a value using i
 return ref i;
}
```

This was a common problem in older C programs, known as a dangling reference. Fortunately, the C# compiler prevents you from committing errors such as this!

## Summary

In this chapter, you learned how to create and use arrays to manipulate sets of data. You saw how to declare and initialize arrays, access data held in arrays, pass arrays as parameters to methods, and return arrays from methods. You also learned how to create multidimensional arrays and how to use arrays of arrays.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 11, “Understanding parameter arrays.”
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

## Quick reference

To	Do this
Declare an array variable	Write the name of the element type, followed by square brackets, followed by the name of the variable, followed by a semicolon. For example: <code>bool[] flags;</code>
Create an instance of an array	Write the keyword <code>new</code> , followed by the name of the element type, followed by the size of the array enclosed in square brackets. For example: <code>bool[] flags = new bool[10];</code>
Initialize the elements of an array to specific values	For an array, write the specific values in a comma-separated list enclosed in braces. For example: <code>bool[] flags = { true, false, true, false };</code>
Find the number of elements in an array	Use the <code>Length</code> property. For example: <code>bool[] flags = ...; ... int numberOfElements = flags.Length;</code>
Access a single array element	Write the name of the array variable, followed by the integer index of the element enclosed in square brackets. Remember, array indexing starts at 0, not 1. For example: <code>bool initialElement = flags[0];</code>
Iterate through the elements of an array	Use a <code>for</code> statement or a <code>foreach</code> statement. For example: <code>bool[] flags = { true, false, true, false }; for (int i = 0; i &lt; flags.Length; i++) {     Console.WriteLine(flags[i]); }  foreach (bool flag in flags) {     Console.WriteLine(flag); }</code>



To	Do this
Declare a multidimensional array variable	Write the name of the element type, followed by a set of square brackets with a comma separator indicating the number of dimensions, followed by the name of the variable, followed by a semicolon. For example, use the following to create a two-dimensional array called <code>table</code> and initialize it to hold four rows of six columns: <pre>int[,] table; table = new int[4,6];</pre>
Declare a jagged array variable	Declare the variable as an array of arrays. You can initialize each nested array to have a different length. For example, use the following to create a jagged array called <code>items</code> and initialize each nested array: <pre>int[][] items; items = new int[4][]; items[0] = new int[3]; items[1] = new int[10]; items[2] = new int[40]; items[3] = new int[25];</pre>

# Understanding parameter arrays

**After completing this chapter, you will be able to:**

- Write a method that can accept any number of arguments by using the `params` keyword.
- Write a method that can accept any number of arguments of any type by using the `params` keyword in combination with the `object` type.
- Explain the differences between methods that take parameter arrays and methods that take optional parameters.

Parameter arrays are useful if you want to write methods that can take any number of arguments, possibly of different types, as parameters. If you're familiar with object-oriented concepts, you might be grinding your teeth in frustration at the previous sentence. After all, the object-oriented approach to solving this problem is to define overloaded methods. However, overloading is not always the most suitable approach. This is especially true if you need to create a method that can take a truly variable number of parameters, each of which might vary in type whenever the method is invoked. This chapter describes how you can use parameter arrays to address situations such as this.

## Overloading: a recap

---

*Overloading* is the technical term for declaring two or more methods with the same name in the same scope. Overloading a method is very useful when you want to perform the same action on arguments of different types.

The classic example of overloading in Microsoft Visual C# is the `Console.WriteLine` method. This method is overloaded numerous times so you can pass any primitive type argument. The following code example illustrates some of the ways in which the `WriteLine` method is defined in the `Console` class:

```
class Console
{
 public static void WriteLine(Int32 value)
 public static void WriteLine(Double value)
 public static void WriteLine(Decimal value)
 public static void WriteLine(Boolean value)
 public static void WriteLine(String value)
 ...
}
```



**Note** The documentation for the `WriteLine` method uses the structure types defined in the `System` namespace for its parameters rather than the C# aliases. For example, the overload that prints out the value for an `int` actually takes an `Int32` as the parameter. Refer to Chapter 9, “Creating value types with enumerations and structures,” for a list of the structure types and their mappings to C# aliases for these types.

As useful as overloading is, it doesn't cover every case. In particular, overloading doesn't easily handle a situation in which the type of parameters doesn't vary, but the number of parameters does. For example, what if you want to write many values to the console? Do you have to provide versions of `Console.WriteLine` that can take two parameters of various combinations, other versions that can take three parameters, and so on? That would quickly become tedious. And wouldn't the massive duplication of these overloaded methods worry you? It should.

Fortunately, there's a way to write a method that takes a variable number of arguments (a *variadic method*): you can use a parameter array, which is declared by using the `params` keyword. To understand how `params` arrays solve this problem, it helps first to understand the uses and shortcomings of ordinary arrays.

## Using array arguments

Suppose you want to write a method to determine the minimum value in a set of values passed as parameters. One way is to use an array. For example, to find the smallest of several `int` values, you could write a static method named `Min` with a single parameter representing an array of `int` values:

```
class Util
{
 public static int Min(int[] paramList)
 {
 // Verify that the caller has provided at least one parameter.
 // If not, throw an ArgumentException exception - it is not possible
 // to find the smallest value in an empty list.
 if (paramList == null || paramList.Length == 0)
 {
 throw new ArgumentException("Utils.Min: not enough arguments");
 }

 // Set the current minimum value found in the list of parameters to the first item.
 int currentMin = paramList[0];

 // Iterate through the list of parameters, searching to see whether any of them
 // are smaller than the value held in currentMin.
 foreach (int i in paramList)
 {
 // If the loop finds an item that is smaller than the value held in
 // currentMin, then set currentMin to this value.
 if (i < currentMin)
 {

```

```

 currentMin = i;
 }
}
// At the end of the loop, currentMin holds the value of the smallest
// item in the list of parameters, so return this value.
return currentMin;
}
}

```



**Note** The `ArgumentException` class is specifically designed to be thrown by a method if the arguments supplied do not meet the requirements of the method.

To use the `Min` method to find the minimum of two `int` variables named `first` and `second`, you can write this:

```

int[] array = new int[2];
array[0] = first;
array[1] = second;
int min = Utils.Min(array);

```

And to use the `Min` method to find the minimum of three `int` variables (named `first`, `second`, and `third`), you can write this:

```

int[] array = new int[3];
array[0] = first;
array[1] = second;
array[2] = third;
int min = Utils.Min(array);

```

You can see that this solution avoids the need for a large number of overloads, but it does so at a price: you must write additional code to populate the array that you pass in. You can, of course, use an anonymous array if you prefer, like this:

```

int min = Utils.Min(new int[] {first, second, third});

```

However, the point is that you still need to create and populate an array, and the syntax can get a little confusing. The solution is to get the compiler to write some of this code for you by using a `params` array as the parameter to the `Min` method.

## Declaring a `params` array

Using a `params` array, you can pass a variable number of arguments to a method. You indicate a `params` array by using the `params` keyword as an array parameter modifier when you define the method parameters. For example, here's `Min` again—this time with its array parameter declared as a `params` array:

```

class Util
{
 public static int Min(params int[] paramList)
 {
 // code exactly as before
 }
}

```

The effect of the `params` keyword on the `Min` method is that it enables you to call the method by using any number of integer arguments without worrying about creating an array. For example, to find the minimum of two integer values, you can simply write this:

```
int min = Utils.Min(first, second);
```

The compiler translates this call into code similar to this:

```
int[] array = new int[2];
array[0] = first;
array[1] = second;
int min = Utils.Min(array);
```

To find the minimum of three integer values, you write the code shown here, which is also converted by the compiler to the corresponding code that uses an array:

```
int min = Utils.Min(first, second, third);
```

Both calls to `Min` (one call with two arguments and the other with three arguments) resolve to the same `Min` method with the `params` keyword. And, as you can probably guess, you can call this `Min` method with any number of `int` arguments. The compiler just counts the number of `int` arguments, creates an `int` array of that size, fills the array with the arguments, and then calls the method by passing the single array parameter.



**Note** If you're a C or C++ programmer, you might recognize `params` as a type-safe equivalent of the `varargs` macros from the `stdarg.h` header file. Java also has a `varargs` facility that operates similarly.

There are several points worth noting about `params` arrays:

- You can't use the `params` keyword with multidimensional arrays. For example, the code in the following example will not compile:

```
// compile-time error
public static int Min(params int[,] table)
...
```

- You can't overload a method based solely on the `params` keyword. The `params` keyword does not form part of a method's signature, as shown in this example. Here, the compiler would not be able to distinguish between these methods in code that calls them:

```
// compile-time error: duplicate declaration
public static int Min(int[] paramList)
...
public static int Min(params int[] paramList)
...
```

- You're not allowed to specify the `ref` or `out` modifier with `params` arrays, as shown in this example:

```
// compile-time errors
public static int Min(ref params int[] paramList)
...
public static int Min(out params int[] paramList)
...
```

- A `params` array must be the last parameter. (This means you can have only one `params` array per method.) Consider this example:

```
// compile-time error
public static int Min(params int[] paramList, int i)
...
```

- A non-`params` method always takes priority over a `params` method. This means you can still create an overloaded version of a method for the common cases, such as in the following example:

```
public static int Min(int leftHandSide, int rightHandSide)
...
public static int Min(params int[] paramList)
...
```

The first version of the `Min` method is used when it's called using two `int` arguments. The second version is used if any other number of `int` arguments is supplied. This includes the case in which the method is called with no arguments. Adding the non-`params` array method might be a useful optimization technique because the compiler won't have to create and populate so many arrays.

## Using `params object[]`

A parameter array of type `int` is very useful. With it, you can pass any number of `int` arguments in a method call. However, what if not only the number of arguments varies, but also the argument type? C# has a way to solve this problem, too. The technique is based on the facts that `object` is the root of all classes and that the compiler can generate code that converts value types (things that aren't classes) to objects by using boxing, as described in Chapter 8, "Understanding values and references."

You can use a parameters array of type `object` to declare a method that accepts any number of `object` arguments, allowing the arguments passed in to be of any type. Look at this example:

```
class Black
{
 public static void Hole(params object[] paramList)
 ...
}
```

I call this method `Black.Hole` because no argument can escape from it:

- You can pass the method no arguments at all, in which case the compiler will pass an object array whose length is 0:

```
Black.Hole(); // converted to Black.Hole(new object[0]);
```

- You can call the `Black.Hole` method by passing `null` as the argument. An array is a reference type, so you're allowed to initialize an array with `null`:

```
Black.Hole(null);
```

- You can pass the `Black.Hole` method an actual array. In other words, you can manually create the array normally generated by the compiler:

```
object[] array = new object[2];
array[0] = "forty two";
array[1] = 42;
Black.Hole(array);
```

- You can pass the `Black.Hole` method arguments of different types, and these arguments will automatically be wrapped inside an object array:

```
Black.Hole("forty two", 42);
//converted to Black.Hole(new object[]{"forty two", 42});
```

## The `Console.WriteLine` method

The `Console` class contains many overloads for the `WriteLine` method. One of these overloads looks like this:

```
public static void WriteLine(string format, params object[] arg);
```

Although string interpolation has very nearly made this version of the `WriteLine` method redundant, this overload was frequently used in previous editions of the C# language. This overload enables the `WriteLine` method to support a format string argument that contains numeric placeholders, each of which can be replaced at runtime with a variable of any type that is specified as a list of parameters (placeholder `{i}` is replaced with the *i*th variable in the list that follows). Here's an example of a call to this method (the variables `fname` and `lname` are strings, `mi` is a `char`, and `age` is an `int`):

```
Console.WriteLine("First name:{0}, Middle Initial:{1}, Last name:{2}, Age:{3}",
 fname, mi, lname, age);
```

The compiler resolves this call into the following:

```
Console.WriteLine("First name:{0}, Middle Initial:{1}, Last name:{2}, Age:{3}",
 new object[4]{fname, mi, lname, age});
```

## Using a params array

In the following exercise, you'll implement and test a static method named `Sum`. The purpose of this method is to calculate the sum of a variable number of `int` arguments passed to it, returning the result as an `int`. You'll do this by writing `Sum` to take a `params int[]` parameter. You'll implement two checks on the `params` parameter to ensure that the `Sum` method is completely robust. You will then call the `Sum` method with a variety of different arguments to test it.

### To write a params array method

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **ParamsArray** solution, which is located in the `\Microsoft Press\VCSBS\Chapter 11\ParamsArray` folder in your **Documents** folder.

The `ParamsArray` project contains the `Program` class in the `Program.cs` file, including the `doWork` method framework you've seen in previous chapters. You'll implement the `Sum` method as a static method of another class called `Utils` (short for utility), which you'll add to the project.

3. In Solution Explorer, right-click the **ParamsArray** project in the `ParamsArray` solution, point to **Add**, and then select **Class**.
4. In the **Add New Item ParamsArray** dialog, in the middle pane, select the **Class** template. In the **Name** box, type **Utils.cs**, and then select **Add**.

The `Utils.cs` file is created and added to the project. It contains an empty class named `Utils` in the `ParamsArray` namespace.

5. Add a public static method named `Sum` to the `Utils` class. This method should return an `int` and accept a `params` array of `int` values named `paramList`. It should look like this:

```
public static int Sum(params int[] paramList)
{
}
```

The first step in implementing the `Sum` method is to check the `paramList` parameter. This parameter can contain a valid set of integers, can be `null`, or can be an array of zero length. In the latter two cases, it's difficult to calculate the sum, so the best option is to throw an `ArgumentException` exception. (You could argue that the sum of the integers in a zero-length array is 0, but you'll treat this situation as an exception in this example.)

6. Add the following code shown in bold to `Sum`. This code throws an `ArgumentException` exception if `paramList` is `null`. The `Sum` method should now look like this:

```
public static int Sum(params int[] paramList)
{
 if (paramList == null)
 {
 throw new ArgumentException("Utils.Sum: null parameter list");
 }
}
```



7. Add code to the `Sum` method to throw an `ArgumentException` exception if the length of the parameter list array is 0, as shown here in bold:

```
public static int Sum(params int[] paramList)
{
 if (paramList is null)
 {
 throw new ArgumentException("Utils.Sum: null parameter list");
 }

 if (paramList.Length == 0)
 {
 throw new ArgumentException("Utils.Sum: empty parameter list");
 }
}
```

If the array passes these two tests, the next step is to add together all the elements inside the array. You can use a `foreach` statement to do this, and you'll need a local variable to hold the running total.

8. Declare an integer variable named `sumTotal` and initialize it to 0, directly after the code from the preceding step.

```
public static int Sum(params int[] paramList)
{
 ...
 if (paramList.Length == 0)
 {
 throw new ArgumentException("Utils.Sum: empty parameter list");
 }

 int sumTotal = 0;
}
```

9. Add a `foreach` statement to the `Sum` method to iterate through the `paramList` array. The body of this `foreach` loop should add each element in the array to `sumTotal`. At the end of the method, return the value of `sumTotal` by using a `return` statement, as shown in bold here:

```
public static int Sum(params int[] paramList)
{
 ...
 int sumTotal = 0;
 foreach (int i in paramList)
 {
 sumTotal += i;
 }
 return sumTotal;
}
```

10. On the **Build** menu, select **Build Solution**, and then confirm that your solution builds without any errors.

## To test the Utils.Sum method

1. Display the Program.cs file in the Code and Text Editor window.
2. In the Code and Text Editor window, delete the `// TODO:` comment and add the following statement to the `doWork` method:

```
Console.WriteLine(Utils.Sum(null));
```

3. On the **Debug** menu, select **Start Without Debugging**.

The program builds and runs, writing the following message to the console:

```
Exception: Utils.Sum: null parameter list
```

This confirms that the first check in the method works.

4. Press **Enter** to close the program and return to Visual Studio 2022.
5. In the Code and Text Editor window, change the call to `Console.WriteLine` in `doWork` as shown here:

```
Console.WriteLine(Utils.Sum());
```

This time, the method is called without any arguments. The compiler translates the empty argument list into an empty array.

6. On the **Debug** menu, select **Start Without Debugging**.

The program builds and runs, writing the following message to the console:

```
Exception: Utils.Sum: empty parameter list
```

This confirms that the second check in the method works.

7. Press **Enter** to close the program and return to Visual Studio 2022.
8. Change the call to `Console.WriteLine` in `doWork` as follows:

```
Console.WriteLine(Utils.Sum(10, 9, 8, 7, 6, 5, 4, 3, 2, 1));
```

9. On the **Debug** menu, select **Start Without Debugging**.

Verify that the program builds, runs, and writes the value 55 to the console.

10. Press **Enter** to close the application and return to Visual Studio 2022.

## Comparing parameter arrays and optional parameters

Chapter 3, “Writing methods and applying scope,” illustrated how to define methods that take optional parameters. At first glance, it appears there is a degree of overlap between methods that use parameter arrays and methods that take optional parameters. However, there are fundamental differences between them:

- A method that takes optional parameters still has a fixed parameter list, and you cannot pass an arbitrary list of arguments. The compiler generates code that inserts the default values onto the stack for any missing arguments before the method runs, and the method is not aware of which of the arguments are provided by the caller and which are compiler-generated defaults.
- A method that uses a parameter array effectively has a completely arbitrary list of parameters, and none of them has a default value. Furthermore, the method can determine exactly how many arguments the caller provided.

Generally, you use parameter arrays for methods that can take any number of parameters (including none), whereas you use optional parameters only where it is not convenient to force a caller to provide an argument for every parameter.

There’s one further situation worth pondering. If you define a method that takes a parameter list and provide an overload that takes optional parameters, it’s not always immediately apparent which version of the method will be called if the argument list in the calling statement matches both method signatures. You’ll investigate this scenario in the final exercise in this chapter.

### To compare a params array and optional parameters

1. Return to the ParamsArray solution in Visual Studio 2022 and display the Utils.cs file in the Code and Text Editor window
2. Add the following `Console.WriteLine` statement shown in bold to the start of the `Sum` method in the `Utils` class:

```
public static int Sum(params int[] paramList)
{
 Console.WriteLine("Using parameter list");
 ...
}
```

3. Add another implementation of the `Sum` method to the `Utils` class. This version should take four optional `int` parameters, each with a default value of 0. In the body of the method, output the message "Using optional parameters". Then calculate and return the sum of the four parameters. The completed method should look like the following code in bold:

```
class Util
{
 ...
 public static int Sum(int param1 = 0, int param2 = 0, int param3 = 0, int param4 = 0)
 {
 Console.WriteLine("Using optional parameters");
 int sumTotal = param1 + param2 + param3 + param4;
 return sumTotal;
 }
}
```

4. Display the `Program.cs` file in the Code and Text Editor window.
5. In the `doWork` method, comment out the existing code and then add the following statement:

```
Console.WriteLine(Utils.Sum(2, 4, 6, 8));
```

This statement calls the `Sum` method, passing four `int` parameters. This call matches both overloads of the `Sum` method.

6. On the **Debug** menu, select **Start Without Debugging** to build and run the application.

When the application runs, it displays the following messages:

```
Using optional parameters
20
```

In this case, the compiler generated code that called the method that takes four optional parameters. This is the version of the method that most closely matches the method call.

7. Press **Enter** and return to Visual Studio.
8. In the `doWork` method, change the statement that calls the `Sum` method and remove the final argument (8), as shown here:

```
Console.WriteLine(Utils.Sum(2, 4, 6));
```

9. On the **Debug** menu, select **Start Without Debugging** to build and run the application.

When the application runs, it displays the following messages:

```
Using optional parameters
12
```

The compiler still generated code that called the method that takes optional parameters, even though the method signature does not exactly match the call. Given a choice between a method that takes optional parameters and a method that takes a parameter list, the C# compiler will use the method that takes optional parameters.

10. Press **Enter** and return to Visual Studio.
11. In the `doWork` method, change the statement that calls the `Sum` method again and add two more arguments (in bold):
- ```
Console.WriteLine(Utils.Sum(2, 4, 6, 8, 10));
```

12. On the **Debug** menu, select **Start Without Debugging** to build and run the application.

When the application runs, it displays the following messages:

```
Using parameter list  
30
```

This time, more arguments are provided than the method that takes optional parameters specifies, so the compiler generated code that calls the method that takes a parameter array.

13. Press **Enter** and return to Visual Studio.

Summary

In this chapter, you learned how to use a `params` array to define a method that can take any number of arguments. You also saw how to use a `params` array of `object` types to create a method that accepts any number of arguments of any type. Finally, you saw how the compiler resolves method calls when it has a choice between calling a method that takes a parameter array and a method that takes optional parameters.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 12, “Working with inheritance.”
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

| To | Do this |
|---|--|
| Write a method that accepts any number of arguments of a given type | Write a method whose parameter is a <code>params</code> array of the given type. For example, a method that accepts any number of <code>bool</code> arguments is declared like this:
<pre>someType Method(params bool[] flags)
{
 ..
}</pre> |
| Write a method that accepts any number of arguments of any type | Write a method whose parameter is a <code>params</code> array whose elements are of type <code>object</code> . For example:
<pre>someType Method(params object[] paramList)
{
 ...
}</pre> |

Working with inheritance

After completing this chapter, you will be able to:

- Understand what inheritance is and how it works.
- Create a derived class that inherits features from a base class.
- Control method hiding and overriding by using the `new`, `virtual`, and `override` keywords.
- Limit accessibility within an inheritance hierarchy by using the `protected` keyword.
- Define extension methods as an alternative mechanism to using inheritance.

Inheritance is a key concept in the world of object-oriented programming. You can use inheritance as a tool to avoid repetition when defining different classes that have some features in common and are quite clearly related to one another.

Suppose you're writing an application to simulate a factory, and you want to specify that managers and manual workers have some similar features but others that differ. For example, both have employee reference numbers assigned to them, but each perform different tasks.

This is where inheritance proves useful. With inheritance, you could create three different classes—managers, manual workers, and all employees—with the managers and manual workers classes inheriting some features from the all employees class. That's what this chapter is about.

What is inheritance?

If you ask several experienced programmers the meaning of the term *inheritance*, you'll typically get different and conflicting answers. Part of the confusion stems from the fact that the word *inheritance* itself has several subtly different meanings. If someone bequeaths something to you in a will, you are said to inherit it. Similarly, we say that you inherit half of your genes from your mother and half of your genes from your father. Both of these uses of the word have very little to do with inheritance in programming.

Inheritance in programming is all about classifying classes, or more precisely, the relationship between classes. For example, when you were at school, you probably learned about mammals, and that horses and whales are examples of mammals. Each has every attribute that a mammal does (it breathes air, it suckles its young, it is warm-blooded, and so on), but each also has its own special features (a horse has hooves, but a whale has flippers and a fluke).

How can you model a horse and a whale in a program? One way is to create two distinct classes named `Horse` and `Whale`. Each class can implement the behaviors that are unique to that type of mammal, such as `Trot` (for a horse) or `Swim` (for a whale), in its own way. But what about behaviors that are common to a horse and a whale, such as `Breathe` and `SuckleYoung`? Although you could add duplicate methods with these names to both classes, this situation becomes a maintenance nightmare, especially if you also decide to start modeling other types of mammals, such as `Human` and `Aardvark`.

In C#, you can use class inheritance to address these issues. A horse, a whale, a human, and an aardvark are all types of mammals, so you can create a class named `Mammal` that provides the common functionality exhibited by these types. You can then declare that the `Horse`, `Whale`, `Human`, and `Aardvark` classes all inherit from `Mammal`. These classes automatically include the functionality of the `Mammal` class (`Breathe`, `SuckleYoung`, and so on), as well as functionality unique to a particular type of mammal (for example, the `Trot` method for the `Horse` class and the `Swim` method for the `Whale` class). Then, if you need to modify how a common method such as `Breathe` works, you need to change it in only one place: the `Mammal` class.

Using inheritance

You declare that a class inherits from another class by using the following syntax:

```
class DerivedClass : BaseClass
{
    ...
}
```

The derived class inherits from the base class, and the methods in the base class become part of the derived class.

In C#, a class can derive from at most one base class; a class *cannot* derive from two or more classes. However, unless `DerivedClass` is declared as `sealed`, you can use the same syntax to derive other classes that inherit from `DerivedClass`. (You'll learn about sealed classes in Chapter 13, "Creating interfaces and defining abstract classes.")

```
class DerivedSubClass : DerivedClass
{
    ...
}
```

Continuing the example described earlier, you could declare the `Mammal` class as follows. The methods `Breathe` and `SuckleYoung` are common to all mammals.

```
class Mammal
{
    public void Breathe()
    {
        ...
    }

    public void SuckleYoung()
    {
        ...
    }
    ...
}
```

You could then define classes for each different type of mammal, adding more methods as necessary, such as in the following example:

```
class Horse : Mammal
{
    ...
    public void Trot()
    {
        ...
    }
}

class Whale : Mammal
{
    ...
    public void Swim()
    {
        ...
    }
}
```



Note If you're a C++ programmer, notice that you do not and cannot explicitly specify whether the inheritance is public, private, or protected. C# inheritance is always implicitly public. If you're familiar with Java, note the use of the colon and that there is no `extends` keyword.

If you create a `Horse` object in your application, you can call the `Trot`, `Breathe`, and `SuckleYoung` methods:

```
Horse myHorse = new Horse();
myHorse.Trot();
myHorse.Breathe();
myHorse.SuckleYoung();
```

Similarly, you can create a `Whale` object, but this time you can call the `Swim`, `Breathe`, and `SuckleYoung` methods; `Trot` is not available because it is defined only in the `Horse` class.



Important Inheritance applies only to classes, not to structures. You cannot define your own inheritance hierarchy with structures, and you cannot define a structure that derives from a class or another structure.

All structures actually inherit from an abstract class named `System.ValueType`. (Chapter 13 explores abstract classes.) This is purely an implementation detail of the way in which the Microsoft .NET libraries define the common behavior for stack-based value types; you are unlikely to make direct use of `ValueType` in your own applications.

The `System.Object` class revisited

The `System.Object` class is the root class of all classes. All classes implicitly derive from `System.Object`. Consequently, the C# compiler silently rewrites the `Mammal` class as the following code (which you can write explicitly if you really want to):

```
class Mammal : System.Object
{
    ...
}
```

Any methods in the `System.Object` class are automatically passed down the chain of inheritance to classes that derive from `Mammal`, such as `Horse` and `Whale`. In practical terms, this means that all classes that you define automatically inherit all the features of the `System.Object` class. This includes methods such as `ToString` (discussed in Chapter 2, “Working with variables, operators, and expressions”), which is used to convert an object to a string, typically for display purposes.

Calling base-class constructors

In addition to the methods that it inherits, a derived class automatically contains all the fields from the base class. These fields usually require initialization when an object is created. You typically perform this kind of initialization in a constructor. Remember that all classes have at least one constructor. (If you don’t provide one, the compiler generates a default constructor for you.)

It’s good practice for a constructor in a derived class to call the constructor for its base class as part of the initialization, which enables the base-class constructor to perform any additional initialization that it requires. You can specify the `base` keyword to call a base-class constructor when you define a constructor for an inheriting class, as shown in this example:

```
class Mammal // base class
{
    public Mammal(string name) // constructor for base class
    {
        ...
    }
    ...
}
```

```

class Horse : Mammal // derived class
{
    public Horse(string name)
        : base(name) // calls Mammal(name)
    {
        ...
    }
    ...
}

```

If you don't explicitly call a base-class constructor in a derived-class constructor, the compiler attempts to silently insert a call to the base class's default constructor before executing the code in the derived-class constructor. Taking the earlier example, the compiler rewrites this as follows:

```

class Horse : Mammal
{
    public Horse(string name)
    {
        ...
    }
    ...
}

```

as this:

```

class Horse : Mammal
{
    public Horse(string name)
        : base()
    {
        ...
    }
    ...
}

```

This works if `Mammal` has a public default constructor. However, not all classes have a public default constructor. For example, remember that the compiler generates a default constructor only if you don't write any nondefault constructors. In this case, forgetting to call the correct base-class constructor results in a compile-time error.

Assigning classes

Previous examples in this book show how to declare a variable by using a class type and how to use the `new` keyword to create an object. There are also examples of how the type-checking rules of C# prevent you from assigning an object of one type to a variable declared as a different type. For example, given the definitions of the `Mammal`, `Horse`, and `Whale` classes shown here, the code that follows these definitions is illegal:

```

class Mammal
{
    ...
}

```

```

class Horse : Mammal
{
    ...
}

class Whale : Mammal
{
    ...
}
...
Horse myHorse = new Horse(...);
Whale myWhale = myHorse;           // error - different types

```

However, it's possible to refer to an object from a variable of a different type as long as the type used is a class that is higher up the inheritance hierarchy. So, the following statements are legal:

```

Horse myHorse = new Horse(...);
Mammal myMammal = myHorse; // legal, Mammal is the base class of Horse

```

If you think about it in logical terms, all horses are mammals, so you can safely assign an object of type `Horse` to a variable of type `Mammal`. The inheritance hierarchy means that you can think of a `Horse` object simply as a special type of `Mammal` object; it has everything that a `Mammal` object has with a few extra bits defined by any methods and fields you added to the `Horse` class. You can also make a `Mammal` variable refer to a `Whale` object.

There is one significant limitation, however: when referring to a `Horse` or `Whale` object by using a `Mammal` variable, you can access only methods and fields that are defined by the `Mammal` class. Any additional methods defined by the `Horse` or `Whale` class are not visible through the `Mammal` class.

```

Horse myHorse = new Horse(...);
Mammal myMammal = myHorse;
myMammal.Breathe();    // OK - Breathe is part of the Mammal class
myMammal.Trot();        // error - Trot is not part of the Mammal class

```



Note The preceding discussion explains why you can assign almost anything to an object variable. Remember that `object` is an alias for `System.Object`, and all classes inherit from `System.Object`, either directly or indirectly.

Be warned that the converse situation is not true. You cannot unreservedly assign a `Mammal` object to a `Horse` variable:

```

Mammal myMammal = new Mammal(...);
Horse myHorse = myMammal; // error

```

This looks like a strange restriction, but remember that not all `Mammal` objects are `Horse` objects. Some might be `Whale` objects. You can assign a `Mammal` object to a `Horse` object as long as you first check that the `Mammal` object is really a `Horse` object, by using the `as` or `is` operator or by using a cast. (Chapter 7, “Creating and managing classes and objects,” discusses the `is` and `as` operators and casts.)

The code example that follows uses the `as` operator to check that `myMammal` refers to a `Horse` object, and if it does, the assignment to `myHorseAgain` results in `myHorseAgain` referring to the same

Horse object. If `myMammal` refers to some other type of object in the `Mammal` class, the `as` operator returns `null` instead:

```
Horse myHorse = new Horse(...);
Mammal myMammal = myHorse;           // myMammal refers to a Horse
...
Horse myHorseAgain = myMammal as Horse; // OK - myMammal was a Horse
...
Whale myWhale = new Whale(...);
myMammal = myWhale;
...
myHorseAgain = myMammal as Horse;     // returns null - myMammal was a Whale
```

Declaring new methods

One of the hardest tasks in the realm of computer programming is thinking up unique and meaningful names for identifiers. If you're defining a method for a class and that class is part of an inheritance hierarchy, sooner or later you'll try to reuse a name that's already in use by one of the classes further up the hierarchy. If a base class and a derived class happen to declare two methods that have the same signature, you will receive a warning when you compile the application.



Note The method signature refers to the name of the method and the number and types of its parameters but not its return type. Two methods that have the same name and that take the same list of parameters have the same signature, even if they return different types.

A method in a derived class masks (or hides) a method in a base class that has the same signature. For example, if you compile the following code, the compiler generates a warning message informing you that `Horse.Talk` hides the inherited method `Mammal.Talk`:

```
class Mammal
{
    ...
    public void Talk() // assume that all mammals can talk
    {
        ...
    }
}

class Horse : Mammal
{
    ...
    public void Talk() // horses talk in a different way from other mammals!
    {
        ...
    }
}
```

Although your code will compile and run, you should take this warning seriously. If another class derives from `Horse` and calls the `Talk` method, it might expect the method implemented in the `Mammal` class to be called. However, the `Talk` method in the `Horse` class hides the `Talk` method in the `Mammal` class, and the `Horse.Talk` method will be called instead.

Most of the time, such a coincidence is at best a source of confusion, and you should consider renaming methods to avoid clashes. However, if you're sure that you want the two methods to have the same signature, thus hiding the `Mammal.Talk` method, you can silence the warning by using the new keyword, as follows:

```
class Mammal
{
    ...
    public void Talk()
    {
        ...
    }
}

class Horse : Mammal
{
    ...
    new public void Talk()
    {
        ...
    }
}
```

Using the new keyword like this does not change the fact that the two methods are completely unrelated and that hiding still occurs. It just turns off the warning. In effect, the new keyword says, "I know what I'm doing, so stop showing me these warnings."

Declaring virtual methods

Sometimes, you do want to hide how a method is implemented in a base class. As an example, consider the `ToString` method in `System.Object`. The purpose of `ToString` is to convert an object to its string representation. Because this method is very useful, it is a member of the `System.Object` class, thereby automatically providing all classes with a `ToString` method. However, how does the version of `ToString` implemented by `System.Object` know how to convert an instance of a derived class to a string? A derived class might contain any number of fields with interesting values that should be part of the string. The answer is that the implementation of `ToString` in `System.Object` is actually a bit simplistic. All it can do is convert an object to a string that contains the name of its type, such as "Mammal" or "Horse". This is not very useful after all. So why provide a method that's so useless? The answer to this second question requires a bit of detailed thought.

Obviously, `ToString` is a fine idea in concept, and all classes should provide a method that can be used to convert objects to strings for display or debugging purposes. It's only the implementation that requires attention. In fact, you're not expected to call the `ToString` method defined by `System.Object`; it is simply a placeholder. Instead, you might find it more useful to provide your own version of the `ToString` method in each class you define, overriding the default implementation in `System.Object`. The version in `System.Object` is there only as a safety net in case a class does not implement or require its own specific version of the `ToString` method.



Note You should be clear on the difference between overriding a method and hiding a method. *Overriding* a method is a mechanism for providing different implementations of the same method; the methods are all related because they are intended to perform the same task, but in a class-specific manner. *Hiding* a method is a means of replacing one method with another; the methods are usually unrelated and might perform totally different tasks. Overriding a method is a useful programming concept; hiding a method is often an error.

A method that is intended to be overridden is called a *virtual method*. You can mark a method as a virtual method by using the `virtual` keyword. For example, the `ToString` method in the `System.Object` class is defined like this:

```
namespace System
{
    class Object
    {
        public virtual string ToString()
        {
            ...
        }
        ...
    }
    ...
}
```



Note If you have experience developing in Java, you should note that C# methods are not virtual by default.

Declaring override methods

If a base class declares that a method is virtual, a derived class can use the `override` keyword to declare another implementation of that method, as demonstrated here:

```
class Horse : Mammal
{
    ...
    public override string ToString()
    {
        ...
    }
}
```

The new implementation of the method in the derived class can call the original implementation of the method in the base class by using the `base` keyword, like this:

```
public override string ToString()
{
    string temp = base.ToString();
    ...
}
```

There are some important rules you must follow when you declare polymorphic methods (discussed in the forthcoming sidebar “Virtual methods and polymorphism”) by using the `virtual` and `override` keywords:

- A virtual method cannot be private; it is intended to be exposed to other classes through inheritance. Similarly, override methods cannot be private because a class cannot change the protection level of a method that it inherits. However, override methods can have a special form of privacy known as *protected access*, as you will find out in the next section.
- The signatures of the virtual and override methods must be identical; they must have the same name, number, and types of parameters. Also, both methods must return the same type.
- You can only override a virtual method. If the base class method is not virtual and you try to override it, you’ll get a compile-time error. This is sensible; it should be up to the designer of the base class to decide whether its methods can be overridden.
- If the derived class does not declare the method by using the `override` keyword, it does not override the base class method; it hides the method. In other words, it becomes an implementation of a completely different method that happens to have the same name. As before, this will cause a compile-time warning, which you can silence by using the new keyword, as previously described.
- An override method is implicitly virtual and can itself be overridden in a further derived class. However, you’re not allowed to explicitly declare that an override method is virtual by using the `virtual` keyword.

Virtual methods and polymorphism

Using virtual methods, you can call different versions of the same method, based on the object type determined dynamically at runtime. Consider the following examples of classes that define a variation on the `Mammal` hierarchy described earlier:

```
class Mammal
{
    ...
    public virtual string GetTypeName()
    {
        return "This is a mammal" ;
    }
}

class Horse : Mammal
{
    ...
    public override string GetTypeName()
    {
        return "This is a horse";
    }
}
```

```

    }
}

class Whale : Mammal
{
    ...
    public override string GetTypeName()
    {
        return "This is a whale";
    }
}

class Aardvark : Mammal
{
    ...
}

```

There are two things that you should note: first, the `override` keyword used by the `GetTypeName` method in the `Horse` and `Whale` classes, and second, the fact that the `Aardvark` class does not have a `GetTypeName` method.

Now examine the following block of code:

```

Mammal myMammal;
Horse myHorse = new Horse(...);
Whale myWhale = new Whale(...);
Aardvark myAardvark = new Aardvark(...);

myMammal = myHorse;
Console.WriteLine(myMammal.GetTypeName()); // Horse
myMammal = myWhale;
Console.WriteLine(myMammal.GetTypeName()); // Whale
myMammal = myAardvark;
Console.WriteLine(myMammal.GetTypeName()); // Aardvark

```

What will the three different `Console.WriteLine` statements output? At first glance, you would expect them all to print "This is a mammal" because each statement calls the `GetTypeName` method on the `myMammal` variable, which is a `Mammal` object. However, in the first case, you can see that `myMammal` is actually a reference to a `Horse` object. (Remember, you are allowed to assign a `Horse` object to a `Mammal` variable because the `Horse` class inherits from the `Mammal` class.) Because the `GetTypeName` method is defined as virtual, the runtime determines that it should call the `Horse.GetTypeName` method, so the statement actually prints the message "This is a horse". The same logic applies to the second `Console.WriteLine` statement, which outputs the message "This is a whale". The third statement calls `Console.WriteLine` on an `Aardvark` object. However, the `Aardvark` class does not have a `GetTypeName` method, so the default method in the `Mammal` class is called, returning the string "This is a mammal".

This phenomenon of the same statement invoking a different method depending on its context is called *polymorphism*, which literally means "many forms."

Understanding protected access

The `public` and `private` keywords create two extremes of accessibility:

- Public fields and methods of a class are accessible to everyone.
- Private fields and methods of a class are accessible only to the class itself.

These two extremes are sufficient when you consider classes in isolation. However, as all experienced object-oriented programmers know, isolated classes cannot solve complex problems.

Inheritance is a powerful way of connecting classes, and there's clearly a special and close relationship between a derived class and its base class. Frequently, it's useful for a base class to allow derived classes to access some of its members while also hiding these members from classes that aren't part of the inheritance hierarchy. In this situation, you can mark members with the `protected` keyword. It works like this:

- If a class A is derived from class B, it can access the protected class members of class B. In other words, inside the derived class A, a protected member of class B is effectively public.
- If a class A is not derived from class B, it cannot access any protected members of class B. So, within class A, a protected member of class B is effectively private.

C# gives programmers the complete freedom to declare methods and fields as protected. However, most object-oriented programming guidelines recommend that you keep your fields strictly private whenever possible and only relax these restrictions when necessary. Public fields violate encapsulation because all users of the class have direct, unrestricted access to the fields. Protected fields maintain encapsulation for users of a class, for whom the protected fields are inaccessible. However, protected fields still allow encapsulation to be violated by other classes that inherit from the base class.



Note You can access a protected base class member not only in a derived class but also in classes derived from the derived class.

In the following exercise, you'll define a simple class hierarchy for modeling different types of vehicles. You'll define a base class named `Vehicle` and derived classes named `Airplane` and `Car`. You will define common methods named `StartEngine` and `StopEngine` in the `Vehicle` class, and you'll add some methods to both of the derived classes that are specific to those classes. Finally, you'll add a virtual method named `Drive` to the `Vehicle` class and override the default implementation of this method in both of the derived classes.

To create a hierarchy of classes

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **Vehicles** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 12\Vehicles** folder in your **Documents** folder.

The Vehicles project contains the file Program.cs, which defines the Program class with the Main and doWork methods that you have seen in previous exercises.

3. In Solution Explorer, right-click the **Vehicles** project, point to **Add**, and then select **Class**.

The Add New Item – Vehicles dialog opens.

4. In the Add New Item – Vehicles dialog, verify that the Class template is highlighted. In the **Name** box, type **Vehicle.cs**, and then select **Add**.

The file Vehicle.cs is created and added to the project and appears in the Code and Text Editor window. The file contains the definition of an empty class named Vehicle.

5. Add the StartEngine and StopEngine methods to the Vehicle class, as shown next in bold:

```
class Vehicle
{
    public void StartEngine(string noiseToMakeWhenStarting)
    {
        Console.WriteLine($"Starting engine: {noiseToMakeWhenStarting}");
    }

    public void StopEngine(string noiseToMakeWhenStopping)
    {
        Console.WriteLine($"Stopping engine: {noiseToMakeWhenStopping}");
    }
}
```

All classes that derive from the Vehicle class will inherit these methods. The values for the noiseToMakeWhenStarting and noiseToMakeWhenStopping parameters will be different for each type of vehicle and will help you to identify which vehicle is being started and stopped later.

6. On the **Project** menu, select **Add Class**.

The Add New Item – Vehicles dialog opens again.

7. In the **Name** box, type **Airplane.cs**, and then select **Add**.

A new file containing a class named Airplane is added to the project and appears in the Code and Text Editor window.

8. In the Code and Text Editor window, modify the definition of the Airplane class so that it inherits from the Vehicle class, as shown in bold here:

```
class Airplane : Vehicle
{
}
```

9. Add the **TakeOff** and **Land** methods to the **Airplane** class, as shown in bold in the following code:

```
class Airplane : Vehicle
{
    public void TakeOff()
    {
        Console.WriteLine("Taking off");
    }

    public void Land()
    {
        Console.WriteLine("Landing");
    }
}
```

10. On the **Project** menu, select **Add Class**.

The Add New Item – Vehicles dialog opens again.

11. In the **Name** text box, type **Car.cs**, and then select **Add**.

A new file containing a class named **Car** is added to the project and appears in the Code and Text Editor window.

12. In the Code and Text Editor window, modify the definition of the **Car** class so that it derives from the **Vehicle** class, as shown here in bold:

```
class Car : Vehicle
{
}
```

13. Add the **Accelerate** and **Brake** methods to the **Car** class, as shown in bold in the following code:

```
class Car : Vehicle
{
    public void Accelerate()
    {
        Console.WriteLine("Accelerating");
    }

    public void Brake()
    {
        Console.WriteLine("Braking");
    }
}
```

14. Display the **Vehicle.cs** file in the Code and Text Editor window.

15. Add the virtual **Drive** method to the **Vehicle** class, as presented here in bold:

```
class Vehicle
{
    ...
    public virtual void Drive()
    {
        Console.WriteLine("Default implementation of the Drive method");
    }
}
```

16. Display the Program.cs file in the Code and Text Editor window.

17. In the `doWork` method, delete the `// TODO:` comment and add the code shown in bold to create an instance of the `Airplane` class and test its methods by simulating a quick journey by airplane, as follows:

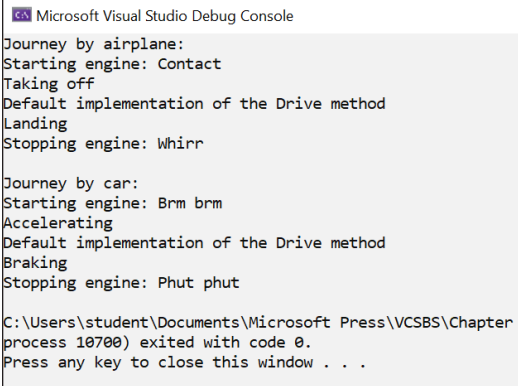
```
static void doWork()
{
    Console.WriteLine("Journey by airplane:");
    Airplane myPlane = new Airplane();
    myPlane.StartEngine("Contact");
    myPlane.TakeOff();
    myPlane.Drive();
    myPlane.Land();
    myPlane.StopEngine("Whirr");
}
```

18. Add the statements that follow (shown in bold) to the `doWork` method after the code you just wrote. These statements create an instance of the `Car` class and test its methods.

```
static void doWork()
{
    ...
    Console.WriteLine();
    Console.WriteLine("Journey by car:");
    Car myCar = new Car();
    myCar.StartEngine("Brm brm");
    myCar.Accelerate();
    myCar.Drive();
    myCar.Brake();
    myCar.StopEngine("Phut phut");
}
```

19. On the **Debug** menu, select **Start Without Debugging**.

In the console window, verify that the program outputs messages simulating the different stages of performing a journey by airplane and by car, as shown in the following image:



```
Microsoft Visual Studio Debug Console
Journey by airplane:
Starting engine: Contact
Taking off
Default implementation of the Drive method
Landing
Stopping engine: Whirr

Journey by car:
Starting engine: Brm brm
Accelerating
Default implementation of the Drive method
Braking
Stopping engine: Phut phut

C:\Users\student\Documents\Microsoft Press\VCSBS\Chapter
process 10700) exited with code 0.
Press any key to close this window . . .
```

Notice that both modes of transport invoke the default implementation of the virtual `Drive` method because neither class currently overrides this method.

20. Press **Enter** to close the application and return to Visual Studio 2022.

21. Display the `Airplane` class in the Code and Text Editor window. Override the `Drive` method in the `Airplane` class as follows in bold:

```
class Airplane : Vehicle
{
    ...
    public override void Drive()
    {
        Console.WriteLine("Flying");
    }
}
```



Note IntelliSense displays a list of available virtual methods. If you select the `Drive` method from the IntelliSense list, Visual Studio automatically inserts a statement in your code that calls the `base.Drive` method. If this happens, delete the statement, because this exercise does not require it.

22. Display the `Car` class in the Code and Text Editor window. Override the `Drive` method in the `Car` class, as shown in bold in the following:

```
class Car : Vehicle
{
    ...
    public override void Drive()
    {
        Console.WriteLine("Motoring");
    }
}
```

23. On the **Debug** menu, select **Start Without Debugging**.

In the console window, the `Airplane` object now displays the message `Flying` when the application calls the `Drive` method, and the `Car` object displays the message `Motoring`:

```
Microsoft Visual Studio Debug Console
Journey by airplane:
Starting engine: Contact
Taking off
Flying
Landing
Stopping engine: Whirr

Journey by car:
Starting engine: Brm brm
Accelerating
Motoring
Braking
Stopping engine: Phut phut

C:\Users\student\Documents\Microsoft Press\VCSBS\Chapter
process 9204) exited with code 0.
Press any key to close this window . . .
```

24. Press **Enter** to close the application and return to Visual Studio 2022.

25. Display the Program.cs file in the Code and Text Editor window.

26. Add the statements shown here in bold to the end of the doWork method:

```
static void doWork()
{
    ...
    Console.WriteLine("\nTesting polymorphism");
    Vehicle v = myCar;
    v.Drive();
    v = myPlane;
    v.Drive();
}
```

This code tests the polymorphism provided by the virtual Drive method. The code creates a reference to the Car object by using a Vehicle variable (which is safe because all Car objects are Vehicle objects) and then calls the Drive method by using this Vehicle variable. The final two statements refer the Vehicle variable to the Airplane object and call what seems to be the same Drive method again.

27. On the **Debug** menu, select **Start Without Debugging**.

28. In the console window, verify that the same messages appear as before, followed by this text:

```
Testing polymorphism
Motoring
Flying
```

The Drive method is virtual, so the runtime (not the compiler) works out which version of the Drive method to call when invoking it through a Vehicle variable, based on the real type of the object referenced by this variable. In the first case, the Vehicle object refers to a Car object, so the application calls the Car.Drive method. In the second case, the Vehicle object refers to an Airplane object, so the application calls the Airplane.Drive method.

29. Press **Enter** to close the application and return to Visual Studio 2022.

Creating extension methods

Inheritance is a powerful feature that makes it possible for you to extend the functionality of a class by creating a new class that derives from it. However, sometimes using inheritance is not the most appropriate mechanism for adding new behaviors, especially if you need to quickly extend a type without affecting existing code.

For example, suppose you want to add a new feature to the int type, such as a method named Negate that returns the negative equivalent value that an integer currently contains. (Of course, you could simply use the unary minus operator [-] to perform the same task, but bear with me....) One way

to achieve this is to define a new type named `NegInt32` that inherits from `System.Int32` (`int` is an alias for `System.Int32`) and adds the `Negate` method:

```
class NegInt32 : System.Int32
{
    public int Negate()
    {
        ...
    }
}
```

The theory is that `NegInt32` will inherit all the functionality associated with the `System.Int32` type in addition to the `Negate` method. There are two reasons why you might not want to follow this approach, however:

- This method applies only to the `NegInt32` type. If you want to use it with existing `int` variables in your code, you must change the definition of every `int` variable to the `NegInt32` type.
- The `System.Int32` type is actually a structure, not a class, and you cannot use inheritance with structures.

This is where extension methods become very useful. Using an extension method, you can extend an existing type (a class or structure) with additional static methods. These static methods become immediately available to your code in any statements that reference data of the type being extended.

You define an extension method in a static class and specify the type to which the method applies as the first parameter to the method, along with the `this` keyword. Here's an example showing how you can implement the `Negate` extension method for the `int` type:

```
static class Utils
{
    public static int Negate(this int i)
    {
        return -i;
    }
}
```

The syntax looks a little odd, but it is the `this` keyword prefixing the parameter to `Negate` that identifies it as an extension method, and the fact that the parameter that `this` prefixes is an `int` means that you're extending the `int` type.

To use the extension method, bring the `Utils` class into scope. (If necessary, add a `using` statement that specifies the namespace to which the `Utils` class belongs, or a `using static` statement that specifies the `Utils` class directly.) Then you can simply use dot notation (`.`) to reference the method, like this:

```
int x = 591;
Console.WriteLine($"x.Negate {x.Negate()}");
```

Notice that you don't need to reference the `Utils` class anywhere in the statement that calls the `Negate` method. The C# compiler automatically detects all extension methods for a given type from all the static classes that are in scope.

You can also invoke the `Utils.Negate` method by passing an `int` as the parameter, using the regular syntax you've seen before, although this use obviates the purpose of defining the method as an extension method:

```
int x = 591; Console.WriteLine($"x.Negate {Utils.Negate(x)}");
```

In the following exercise, you'll add an extension method to the `int` type. With this extension method, you can convert the value in an `int` variable from base 10 to a representation of that value in a different number base.

To create an extension method

1. In Visual Studio 2022, open the **ExtensionMethod** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 12\ExtensionMethod** folder in your **Documents** folder.
2. Display the `Utils.cs` file in the Code and Text Editor window.

This file contains a static class named `Utils` in a namespace named `Extensions`. Remember that you must define extension methods inside a static class. The class is empty apart from the `// TODO: comment`.

3. Delete the comment and declare a public static method in the `Utils` class named `ConvertToBase`. The method should take two parameters:
 - An `int` parameter named `i`, prefixed with the `this` keyword to indicate that the method is an extension method for the `int` type
 - Another ordinary `int` parameter named `baseToConvertTo`

The method will convert the value in `i` to the base indicated by `baseToConvertTo`. The method should return an `int` containing the converted value.

The `ConvertToBase` method should look like this:

```
static class Utils
{
    public static int ConvertToBase(this int i, int baseToConvertTo)
    {
    }
}
```

4. Add an `if` statement to the `ConvertToBase` method that checks that the value of the `baseToConvertTo` parameter is between 2 and 10.

The algorithm used by this exercise does not work reliably outside this range of values. Throw an `ArgumentException` exception with a suitable message if the value of `baseToConvertTo` is outside this range.

The `ConvertToBase` method should look like this:

```
public static int ConvertToBase(this int i, int baseToConvertTo)
{
    if (baseToConvertTo < 2 || baseToConvertTo > 10)
```



```

        {
            throw new ArgumentException("Value cannot be converted to base " +
                baseToConvertTo.ToString());
        }
    }
}

```

5. Add the following statements shown in bold to the `ConvertToBase` method, after the statement block that throws the `ArgumentException` exception.

```

public static int ConvertToBase(this int i, int baseToConvertTo)
{
    ...
    int result = 0;
    int iterations = 0;
    do
    {
        int nextDigit = i % baseToConvertTo;
        i /= baseToConvertTo;
        result += nextDigit * (int)Math.Pow(10, iterations);
        iterations++;
    }
    while (i != 0);
    return result;
}

```

This code implements a well-known algorithm that converts a number from base 10 to a different number base. (Chapter 5, "Using compound assignment and iteration statements," presents a version of this algorithm for converting a decimal number to an octal.)

6. Display the `Program.cs` file in the Code and Text Editor window.
7. Add the following `using` directive after the `using System;` directive at the top of the file:

```
using Extensions;
```

This statement brings the namespace containing the `Utils` class into scope. The compiler might display the warning "`using directive is unnecessary`"; however, the `ConvertToBase` extension method will not be visible in the `Program.cs` file if you don't perform this task.

8. Add the following statements shown in bold to the `doWork` method of the `Program` class, replacing the `// TODO:` comment:

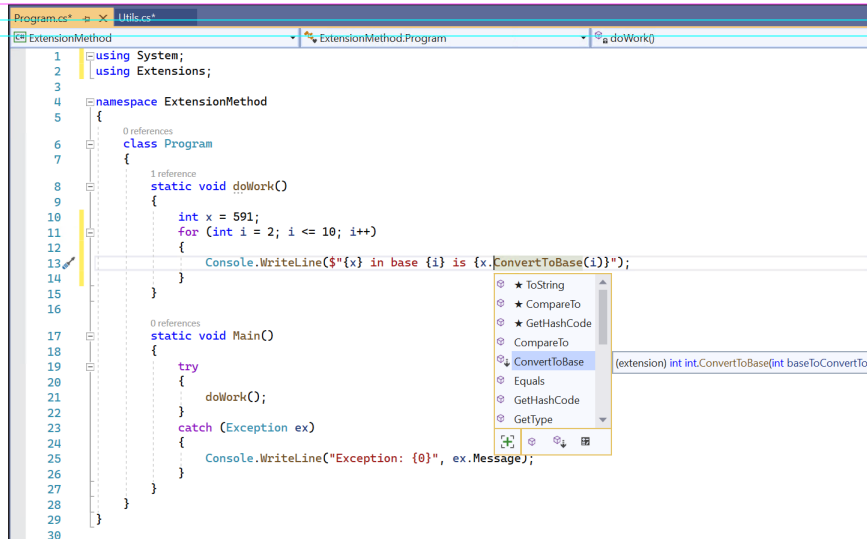
```

static void doWork()
{
    int x = 591;
    for (int i = 2; i <= 10; i++)
    {
        Console.WriteLine($"{x} in base {i} is {x.ConvertToBase(i)}");
    }
}

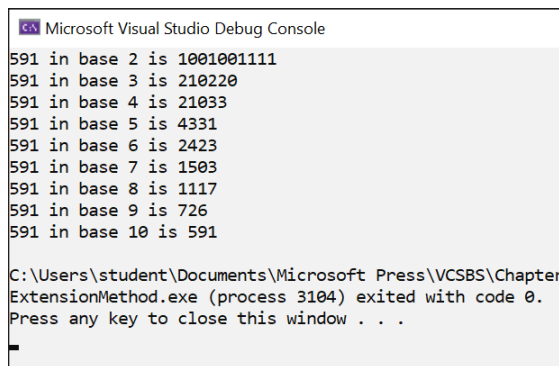
```

This code creates an `int` named `x` and sets it to the value 591. (You can pick any integer value you want.) The code then uses a loop to print out the value 591 in all number bases between 2 and 10.

Notice that `ConvertToBase` appears as an extension method in IntelliSense when you type the period (.) after `x` in the `Console.WriteLine` statement.



9. On the **Debug** menu, select **Start Without Debugging**. Confirm that the program displays messages to the console showing the value 591 in the different number bases, like this:



10. Press **Enter** to close the program and return to Visual Studio 2022.

Summary

In this chapter, you learned how to use inheritance to define a hierarchy of classes, and you should now understand how to override inherited methods and implement virtual methods. You also learned how to add an extension method to an existing type.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 13.
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

| To | Do this |
|--|--|
| Create a derived class from a base class | Declare the new class name followed by a colon and the name of the base class. For example:
<pre>class DerivedClass : BaseClass { ... }</pre> |
| Call a base-class constructor as part of the constructor for an inheriting class | Suffix the definition of the constructor with a call to base, before the body of the derived-class constructor, and provide any necessary parameters to the base constructor. For example:
<pre>class DerivedClass : BaseClass { ... public DerivedClass(int x) : base(x) { ... } ... }</pre> |
| Declare a virtual method | Use the <code>virtual</code> keyword when declaring the method. For example:
<pre>class Mammal { public virtual void Breathe() { ... } ... }</pre> |
| Implement a method in a derived class that overrides an inherited virtual method | Use the <code>override</code> keyword when declaring the method in the derived class. For example:
<pre>class Whale : Mammal { public override void Breathe() { ... } ... }</pre> |
| Define an extension method for a type | Add a static public method to a static class. The first parameter must be of the type being extended, preceded by the <code>this</code> keyword. For example:
<pre>static class Utils { public static int Negate(this int i) { return -i; } }</pre> |

Creating interfaces and defining abstract classes

After completing this chapter, you will be able to:

- Define an interface specifying the signatures and return types of methods.
- Implement an interface in a structure or class.
- Reference a class through an interface.
- Capture common implementation details in an abstract class.
- Implement sealed classes that cannot be used to derive new classes.

Inheriting from a class is a powerful mechanism, but the real power of inheritance comes from inheriting from an interface. An interface does not contain any code or data; it just specifies the methods and properties that a class that inherits from the interface must provide. By using an interface, you can completely separate the names and signatures of the methods of a class from the method's implementation.

Abstract classes are similar to interfaces in many ways, except that abstract classes can contain code and data. However, you can specify certain methods of an abstract class as virtual so that a class that inherits from the abstract class can optionally provide its own implementation of these methods.

Abstract classes are frequently used with interfaces. Together, they provide a key technique with which you can build extensible programming frameworks, as you'll discover in this chapter.

Understanding interfaces

Suppose you want to define a new class in which you can store collections of objects, a bit like you would use an array. However, unlike with an array, you want to provide a method named `RetrieveInOrder` to enable applications to retrieve objects in a sequence that depends on the type of object the collection contains. (With an ordinary array, you can iterate through its contents, and by default, you retrieve items according to their index.) For example, if the collection holds alpha-numeric objects such as strings, the collection should allow an application to retrieve these strings in sequence according to the collating sequence of the computer, and if the collection holds numeric objects such as integers, the collection should enable the application to retrieve objects in numerical order.

When you define the collection class, you don't want to restrict the types of objects that it can hold. (The objects can even be class or structure types.) Consequently, you don't know how to order these objects. So, how do you provide the collection class with a method that sorts objects whose types you do not know when you actually write the collection class?

At first glance, this problem seems similar to the `Tostring` problem described in Chapter 12, "Working with inheritance," which could be resolved by declaring a virtual method that subclasses of your collection class can override. However, any similarity is misleading. There is no inheritance relationship between the collection class and the objects that it holds, so a virtual method would not be of much use.

If you think for a moment, the problem is that the way in which the objects in the collection should be ordered depends on the type of the object in the collection and not on the collection itself. The solution is to require that all the objects provide a method, such as the `CompareTo` method shown in the following example, that the `RetrieveInOrder` method of the collection can call, making it possible for the collection to compare these objects with one another:

```
int CompareTo(object obj)
{
    // return 0 if this instance is equal to obj
    // return < 0 if this instance is less than obj
    // return > 0 if this instance is greater than obj
    ...
}
```

You can define an interface for collectible objects that includes the `CompareTo` method and specify that the collection class can contain only classes that implement this interface. In this way, an interface is similar to a contract. If a class implements an interface, the interface guarantees that the class contains all the methods specified in the interface. This mechanism ensures that you'll be able to call the `CompareTo` method on all objects in the collection and sort them.

Using interfaces, you can truly separate the what from the how. An interface gives you only the name, return type, and parameters of the method. Exactly how the method is implemented is not a concern of the interface. The interface describes the functionality that a class should provide but not how this functionality is implemented.

Defining an interface

Defining an interface is syntactically similar to defining a class, except that you use the `interface` keyword instead of the `class` keyword. Within the interface, you declare methods exactly as in a class or structure, except that you never specify an access modifier (`public`, `private`, or `protected`). Additionally, the methods in an interface have no implementation; they're simply declarations, and all types that implement the interface must provide their own implementations. Consequently, you replace the method body with a semicolon. Here's an example:

```
interface IComparable
{
    int CompareTo(object obj);
}
```



Tip The Microsoft .NET documentation recommends that you preface the name of your interfaces with the capital letter `I`. This convention is the last vestige of Hungarian notation in C#. Incidentally, the `System` namespace already defines the `IComparable` interface as just shown.



Important An interface cannot contain any data; you cannot add fields (not even private ones) to an interface.

Implementing an interface

To implement an interface, you declare a class or structure that inherits from the interface and that implements all the methods specified by the interface. This is not really inheritance as such, although the syntax is the same and some of the semantics that you will see later in this chapter bear many of the hallmarks of inheritance.



Note Unlike with class inheritance, a structure can implement an interface.

For example, suppose you're defining the `Mammal` hierarchy described in Chapter 12, but you need to specify that land-bound mammals provide a method named `NumberOfLegs` that returns as an `int` the number of legs that a mammal has. (Sea-bound mammals do not implement this interface.) You could define the `ILandBound` interface that contains this method as follows:

```
interface ILandBound
{
    int NumberOfLegs();
}
```

You could then implement this interface in the `Horse` class. You inherit from the interface and provide an implementation of every method defined by the interface (in this case, there's just the one method, `NumberOfLegs`).

```
class Horse : ILandBound
{
    ...
    public int NumberOfLegs()
    {
        return 4;
    }
}
```

When you implement an interface, you must ensure that each method matches its corresponding interface method exactly, according to the following rules:

- The method names and return types match exactly.
- Any parameters (including `ref` and `out` keyword modifiers) match exactly.

- All methods implementing an interface must be publicly accessible. However, if you're using an explicit interface implementation, the method shouldn't have an access qualifier.

If there's any difference between the interface definition and its declared implementation, the class will not compile.



Tip The Microsoft Visual Studio integrated development environment (IDE) can help reduce coding errors caused by failing to implement the methods in an interface. The Implement Interface wizard can generate stubs for each item in an interface that a class implements. You then fill in these stubs with the appropriate code. You'll see how to use this wizard in the exercises later in this chapter.

A class can inherit from another class and implement an interface at the same time. In this case, C# does not distinguish between the base class and the interface by using specific keywords as, for example, Java does. Instead, C# uses a positional notation. The base class is always named first, followed by a comma, followed by the interface. The following example defines `Horse` as a class that is a `Mammal` but that additionally implements the `ILandBound` interface:

```
interface ILandBound
{
    ...
}

class Mammal
{
    ...
}

class Horse : Mammal , ILandBound
{
    ...
}
```



Note An interface, such as `InterfaceA`, can inherit from another interface, `InterfaceB`. Technically, this is known as *interface extension* rather than inheritance. In this case, any class or struct that implements `InterfaceA` must provide implementations of all the methods in `InterfaceB` and `InterfaceA`.

Referencing a class through its interface

In the same way that you can reference an object by using a variable defined as a class that is higher up the hierarchy, you can reference an object by using a variable defined as an interface that the object's class implements. Taking the preceding example, you can reference a `Horse` object by using an `ILandBound` variable, as follows:

```
Horse myHorse = new Horse(...);
ILandBound iMyHorse = myHorse; // legal
```

This works because all horses are land-bound mammals. The converse is not true, however; you cannot assign an `ILandBound` object to a `Horse` variable without casting it first to verify that it does actually reference a `Horse` object and not some other class that also happens to implement the `ILandBound` interface.

The technique of referencing an object through an interface is useful because you can employ it to define methods that can take different types as parameters—as long as the types implement a specified interface. For example, the `FindLandSpeed` method shown here can take any argument that implements the `ILandBound` interface:

```
int FindLandSpeed(ILandBound landBoundMammal)
{
    ...
}
```

You can verify that an object is an instance of a class that implements a specific interface by using the `is` operator, which is demonstrated in Chapter 8, “Understanding values and references.” You use the `is` operator to determine whether an object has a specified type, and it works with interfaces as well as with classes and structs. For example, the following block of code checks that the variable `myHorse` actually implements the `ILandBound` interface before attempting to assign it to an `ILandBound` variable:

```
if (myHorse is ILandBound)
{
    ILandBound iLandBoundAnimal = myHorse;
}
```



Note When referencing an object through an interface, you can invoke only methods that are visible through the interface.

Working with multiple interfaces

A class can have at most one base class, but it’s allowed to implement an unlimited number of interfaces. A class must implement all the methods declared by these interfaces.

If a structure or class implements more than one interface, you specify the interfaces as a comma-separated list. If a class also has a base class, the interfaces are listed *after* the base class. For example, suppose you define another interface named `IGrazable` that contains the `ChewGrass` method for all grazing animals. You can define the `Horse` class like this:

```
class Horse : Mammal, ILandBound, IGrazable
{
    ...
}
```


Explicitly implementing an interface

The examples so far have shown classes that implicitly implement an interface. If you revisit the `ILandBound` interface and the `Horse` class (shown next), you'll see that although the `Horse` class implements from the `ILandBound` interface, nothing in the implementation of the `NumberOfLegs` method in the `Horse` class says that it is part of the `ILandBound` interface:

```
interface ILandBound
{
    int NumberOfLegs();
}

class Horse : ILandBound
{
    ...
    public int NumberOfLegs()
    {
        return 4;
    }
}
```

This might not be an issue in a simple situation, but suppose the `Horse` class implemented multiple interfaces. There's nothing to prevent multiple interfaces from specifying a method with the same name, although they might have different semantics. For example, suppose you want to implement a transportation system based on horse-drawn coaches. A lengthy journey might be broken down into several stages, or legs. If you want to keep track of how many legs each horse had pulled the coach for, you might define the following interface:

```
interface IJourney
{
    int NumberOfLegs();
}
```

Now, if you implement this interface in the `Horse` class, you have an interesting problem:

```
class Horse : ILandBound, IJourney
{
    ...
    public int NumberOfLegs()
    {
        return 4;
    }
}
```

This is legal code, but does the horse have four legs or has it pulled the coach for four legs of the journey? The answer as far as C# is concerned is both! By default, C# does not distinguish which interface the method is implementing, so the same method actually implements both interfaces.

To solve this problem and disambiguate which method is part of which interface implementation, you can implement interfaces explicitly. To do this, you specify which interface a method belongs to when you implement it, like this:

```
class Horse : ILandBound, IJourney
{
    ...
    int ILandBound.NumberOfLegs()
    {
        return 4;
    }

    int IJourney.NumberOfLegs()
    {
        return 3;
    }
}
```

Now you can see that the horse has four legs and has pulled the coach for three legs of the journey.

Apart from prefixing the name of the method with the interface name, there's one other subtle difference in this syntax: the methods are not marked public. You cannot specify the protection for methods that are part of an explicit interface implementation. This leads to another interesting phenomenon. If you create a Horse variable in the code, you cannot actually invoke either of the NumberOfLegs methods because they are not visible. As far as the Horse class is concerned, they are both private. In fact, this makes sense. If the methods were visible through the Horse class, which method would the following code actually invoke, the one for the ILandBound interface or the one for the IJourney interface?

```
Horse horse = new Horse();
...
// The following statement will not compile
int legs = horse.NumberOfLegs();
```

So, how do you access these methods? The answer is that you reference the Horse object through the appropriate interface, like this:

```
Horse horse = new Horse();
...
IJourney journeyHorse = horse;
int legsInJourney = journeyHorse.NumberOfLegs();
ILandBound landBoundHorse = horse;
int legsOnHorse = landBoundHorse.NumberOfLegs();
```



Tip I recommend explicitly implementing interfaces when possible.

Handling versioning with interfaces

An interface is an excellent tool for defining the *shape* of a class, enabling multiple developers to collaborate on complex projects more easily, and adding extensibility to a system. For example, once a developer has stipulated the methods and operations available through an interface, other developers can take this definition as a specification and use it in their apps without worrying about how the classes that implement this interface actually work. Additionally, multiple types might implement the same interface. These types essentially become interchangeable. An app that invokes methods through an interface can instantiate any type that implements this interface and expect it to work seamlessly.

However, historically, interfaces were not very good at handling versioning. An interface was considered as an immutable definition of a type. Once an interface had been agreed upon, it became difficult to modify without requiring that all apps that reference the interface also be modified. Consider the following scenario.

You define the following simple interface:

```
public interface IMyInterface
{
    public void DoSomeWork();
}
```

Later, another developer creates a class that implements this same interface:

```
public class myClass : IMyInterface
{
    void IMyInterface.DoSomeWork()
    {
        // Implementation
        ...
    }
}
```

Sometime later, you decide to extend the interface and add another method:

```
public interface IMyInterface
{
    public void DoSomeWork();
    public void DoAdditionalWork();
}
```

This change will break all existing apps that use the original interface because they don't implement the `DoAdditionalWork` method.

The traditional way to solve this problem is to create a new interface that extends the existing interface, like this:

```
public interface IMyExtendedInterface : IMyInterface
{
    public void DoAdditionalWork();
}
```

Existing apps that use the original interface will continue to function correctly, while new apps can implement `myExtendedInterface`.

Unfortunately, this approach can get messy as the interfaces evolve over time, resulting in a complex chain of interfaces that can be difficult to track and maintain. Visual C# 9 addresses this problem by enabling you to provide a default implementation of a method in an interface, like this:

```
public interface IMyInterface
{
    public void DoSomeWork();
    public void DoAdditionalWork()
    {
        throw new NotImplementedException();
    }
}
```

Any type that implements the interface but doesn't create its own version of the `DoAdditionalWork` method will automatically use this default implementation.

You can perform any C# operations you like in a default method implementation; it is a regular C# method. However, my preference is to simply throw a `NotImplementedException` exception, as shown previously. Existing apps won't call this method, and any new apps that use the extended version of the interface should provide their own implementation of this method. To do this, a class simply creates a method with the same signature. Don't use the `override` keyword as you would when inheriting from a class:

```
public class myNewClass : IMyInterface
{
    void IMyInterface.DoSomeWork()
    {
        // Implementation
        ...
    }

    void IMyInterface.DoAdditionalWork()
    {
        // Implementation
        ...
    }
}
```

Interface restrictions

There's another reason for keeping the default implementation of a method simple. Essentially, an interface is not a class, and you shouldn't treat it as such. Remember that an interface is a definition of the *shape* of a type, rather than an implementation of the type. The following restrictions are natural consequences of this:

- You're not allowed to define any fields in an interface, not even static fields. A field is an implementation detail of a class or structure.
- You're not allowed to define any constructors in an interface. A constructor is also considered to be an implementation detail of a class or structure.

- You're not allowed to define a destructor in an interface. A destructor contains the statements used to destroy an object instance. (Destructors are described in Chapter 14, "Using garbage collection and resource management.")
- You cannot specify an access modifier for any method. All methods in an interface are implicitly public.
- You cannot nest any types (such as enumerations, structures, classes, or interfaces) inside an interface.
- An interface is not allowed to inherit from a structure, record, or class, although an interface can inherit from another interface. Structures, records, and classes contain implementation details; if an interface were allowed to inherit from any of these types, it would be inheriting some implementation details.

Defining and using interfaces

In the following exercises, you'll define and implement interfaces that constitute part of a simple graphical drawing package. You will define two interfaces, called `IDraw` and `IColor`, and then you'll define classes that implement them. Each class will define a shape that can be drawn on a canvas on a form. (A *canvas* is a control that you can use to draw lines, text, and shapes on the screen.)

The `IDraw` interface defines the following methods:

- **SetLocation** With this method, you can specify the position of the top-left corner as x- and y-coordinates of the shape on the canvas.
- **Draw** This method actually draws the shape on the canvas at the location specified by using the `SetLocation` method.

The `IColor` interface defines the following method:

- **SetColor** You use this method to specify the color of the shape. When the shape is drawn on the canvas, it will appear in this color.

To define the `IDraw` and `IColor` interfaces

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **Drawing** solution, which is located in the `\Microsoft Press\VCSBS\Chapter 13\Drawing` folder in your **Documents** folder.

The **Drawing** project is a graphical application. It contains a form called `DrawingPad`. This form contains a canvas control called `drawingCanvas`. You will use this form and canvas to test your code.

3. In Solution Explorer, select the **Drawing** project. On the **Project** menu, select **Add New Item**.

4. In the left pane of the Add New Item – Drawing dialog, select **Visual C#**, and then select **Code**. In the middle pane, select the **Interface** template. In the **Name** box, type **IDraw.cs**, and then select **Add**.

Visual Studio creates the IDraw.cs file and adds it to your project. The IDraw.cs file appears in the Code and Text Editor window, and should look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Drawing
{
    interface IDraw
    {
    }
}
```

5. In the IDraw.cs file, add the following `using` directive to the list at the top of the file:

```
using Windows.UI.Xaml.Controls;
```

You'll reference the Canvas class in this interface. The Canvas class is located in the `Windows.UI.Xaml.Controls` namespace for Universal Windows Platform (UWP) apps.

6. Add the methods shown here in bold to the IDraw interface:

```
interface IDraw
{
    void SetLocation(int xCoord, int yCoord);
    void Draw(Canvas canvas);
}
```

7. On the **Project** menu, select **Add New Item** again.
8. In the Add New Item – Drawing dialog, in the middle pane, select the **Interface** template. In the **Name** box, type **IColor.cs**, and then select **Add**.

Visual Studio creates the IColor.cs file and adds it to your project. The IColor.cs file appears in the Code and Text Editor window.

9. In the IColor.cs file, at the top of the file, add the following `using` directive to the list:

```
using Windows.UI;
```

You'll reference the Color class in this interface, which is located in the `Windows.UI` namespace for UWP apps.

10. Add the following method shown in bold to the IColor interface definition:

```
interface IColor
{
    void SetColor(Color color);
}
```

You've now defined the `IDraw` and `IColor` interfaces. The next step is to create some classes that implement them. In the following exercise, you'll create two new shape classes, called `Square` and `Circle`. These classes will implement both interfaces.

To create the `Square` and `Circle` classes and implement the interfaces

1. On the **Project** menu, select **Add Class**.
2. In the Add New Item – Drawing dialog, in the middle pane, verify that the **Class** template is selected. In the **Name** box, type **Square.cs**, and then select **Add**.

Visual Studio creates the `Square.cs` file and displays it in the Code and Text Editor window.

3. At the top of the `Square.cs` file, add the following `using` directives to the list:

```
using Windows.UI;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Shapes;
using Windows.UI.Xaml.Controls;
```

4. Modify the definition of the `Square` class so that it implements the `IDraw` and `IColor` interfaces, as shown here in bold:

```
class Square: IDraw, IColor
{
}
```

5. Add the following private variables shown in bold to the `Square` class:

```
class Square : IDraw, IColor
{
    private int sideLength;
    private int locX = 0, locY = 0;
    private Rectangle rect = null;
}
```

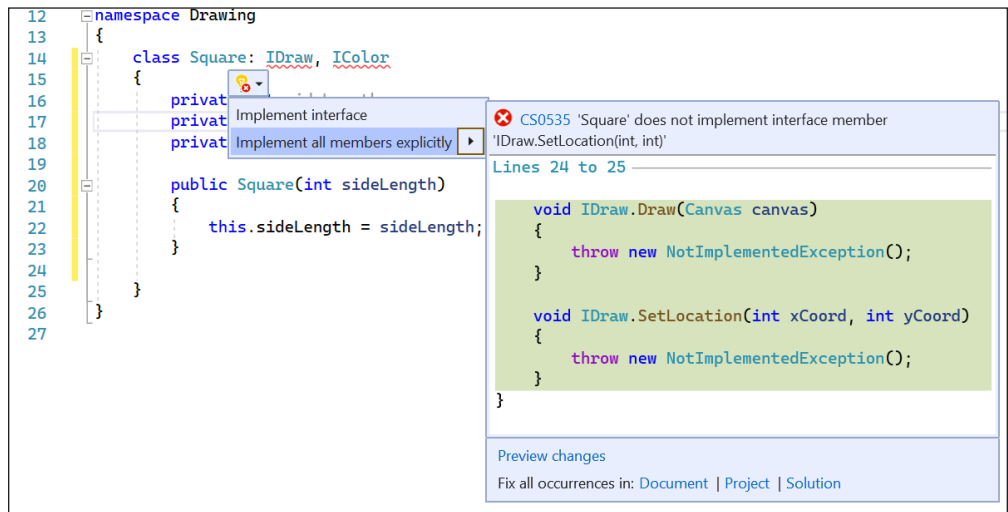
These variables will hold the position and size of the `Square` object on the canvas. The `Rectangle` class is located in the `Windows.UI.Xaml.Shapes` namespace for UWP apps. You'll use this class to draw the square.

6. Add the following constructor shown in bold to the `Square` class:

```
class Square : IDraw, IColor
{
    ...
    public Square(int sideLength)
    {
        this.sideLength = sideLength;
    }
}
```

This constructor initializes the `sideLength` field and specifies the length of each side of the square.

7. In the definition of the `Square` class, hover over the `IDraw` interface. Select the lightbulb context menu that appears, and then select **Implement All Members Explicitly**, as shown here:



This feature causes Visual Studio to generate default implementations of the methods in the `IDraw` interface. You can also add the methods to the `Square` class manually if you prefer. The following example shows the code generated by Visual Studio:

```
void IDraw.Draw(Canvas canvas)
{
    throw new NotImplementedException();
}

void IDraw.SetLocation(int xCoord, int yCoord)
{
    throw new NotImplementedException();
}
```

Each of these methods currently throws a `NotImplementedException` exception. You are expected to replace the body of these methods with your own code.

8. In the `IDraw.SetLocation` method, replace the existing code that throws a `NotImplementedException` exception with the following statements shown in bold:

```
void IDraw.SetLocation(int xCoord, int yCoord)
{
    this.locX = xCoord;
    this.locY = yCoord;
}
```

This code stores the values passed in through the parameters in the `locX` and `locY` fields in the `Square` object.

9. Replace the exception code generated in the `IDraw.Draw` method with the statements shown here in bold:

```
void IDraw.Draw(Canvas canvas)
{
    this.rect = new Rectangle();
    this.rect.Height = this.sideLength;
    this.rect.Width = this.sideLength;
    Canvas.SetTop(this.rect, this.locY);
    Canvas.SetLeft(this.rect, this.locX);
    canvas.Children.Add(this.rect);
}
```

This method renders the `Square` object by drawing a `Rectangle` shape on the canvas. (A square is simply a rectangle for which all four sides have the same length.) The height and width of the `Rectangle` are set by using the value of the `sideLength` field. The position of the `Rectangle` shape on the canvas is set by using the static `SetTop` and `SetLeft` methods of the `Canvas` class, and then the `Rectangle` shape is added to the canvas. (This causes it to be displayed.)

10. Add the `SetColor` method from the `IColor` interface to the `Square` class, as shown here:

```
void IColor.SetColor(Color color)
{
    if (this.rect is not null)
    {
        SolidColorBrush brush = new SolidColorBrush(color);
        this.rect.Fill = brush;
    }
}
```

This method checks that the `Square` object has actually been displayed. (The `rect` field will be `null` if it has not yet been rendered.) The code sets the `Fill` property of the `rect` field with the specified color by using a `SolidColorBrush` object. (The details of how the `SolidColorBrush` class works are beyond the scope of this discussion.)

11. On the **Project** menu, select **Add Class**.
12. In the Add New Item – Drawing dialog, in the **Name** box, type **Circle.cs**, and then select **Add**.

Visual Studio creates the `Circle.cs` file and displays it in the Code and Text Editor window.

13. At the top of the `Circle.cs` file, add the following `using` directives to the list:

```
using Windows.UI;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Shapes;
using Windows.UI.Xaml.Controls;
```

14. Modify the definition of the `Circle` class so that it implements the `IDraw` and `IColor` interfaces as shown here in bold:

```
class Circle : IDraw, IColor
{
}
```

- 15.** Add the following private variables shown in bold to the `Circle` class:

```
class Circle : IDraw, IColor
{
    private int diameter;
    private int locX = 0, locY = 0;
    private Ellipse circle = null;
}
```

These variables will hold the position and size of the `Circle` object on the canvas. The `Ellipse` class provides the functionality that you will use to draw the circle.

- 16.** Add the constructor shown here in bold to the `Circle` class:

```
class Circle : IDraw, IColor
{
    ...
    public Circle(int diameter)
    {
        this.diameter = diameter;
    }
}
```

This constructor initializes the `diameter` field.

- 17.** Add the following `SetLocation` method from the `IDraw` interface to the `Circle` class:

```
void IDraw.SetLocation(int xCoord, int yCoord)
{
    this.locX = xCoord;
    this.locY = yCoord;
}
```



Note This method is the same as the one for the `Square` class. You'll see how you refactor the code to avoid this repetition later in this chapter.

- 18.** Add the `Draw` method shown here to the `Circle` class:

```
void IDraw.Draw(Canvas canvas)
{
    this.circle = new Ellipse();
    this.circle.Height = this.diameter;
    this.circle.Width = this.diameter;
    Canvas.SetTop(this.circle, this.locY);
    Canvas.SetLeft(this.circle, this.locX);
    canvas.Children.Add(this.circle);
}
```

This method is also part of the `IDraw` interface. It's similar to the `Draw` method in the `Square` class, except it renders the `Circle` object by drawing an `Ellipse` shape on the canvas. (A circle is an ellipse for which the width and height are the same.) As with the `SetLocation` method, you'll see how to refactor this code to reduce any repetition later in this chapter.

19. Add the following `SetColor` method to the `Circle` class:

```
void IColor.SetColor(Color color)
{
    if (this.circle is not null)
    {
        SolidColorBrush brush = new SolidColorBrush(color);
        this.circle.Fill = brush;
    }
}
```

This method is part of the `IColor` interface. This method is similar to that of the `Square` class.

You have completed the `Square` and `Circle` classes. You can now use the form to test them.

To test the `Square` and `Circle` classes

1. Display the `DrawingPad.xaml` file in the Design View window.
2. On the form, select the large shaded area.

The shaded area of the form is the `Canvas` object. You select this area to set the focus to this object.

3. In the Properties window, select the **Event Handlers** button. (This button has an icon that looks like a bolt of lightning.)
4. In the list of events, locate the `Tapped` event, and then double-click the **Tapped** box.

Visual Studio creates a method called `drawingCanvas_Tapped` for the `DrawingPad` class and displays it in the Code and Text Editor window. This is an event handler that runs when the user taps the canvas with a finger or clicks the left mouse button on the canvas. You can learn more about event handlers in Chapter 20, “Decoupling application logic and handling events.”

5. At the top of the `DrawingPad.xaml.cs` file, add the following `using` directive to the list:

```
using Windows.UI;
```

The `Windows.UI` namespace contains the definition of the `Colors` class, which you'll use when you set the color of a shape as it is drawn.

6. Add the following code shown in bold to the `drawingCanvas_Tapped` method:

```
private void drawingCanvas_Tapped(object sender, TappedRoutedEventArgs e)
{
    Point mouseLocation = e.GetPosition(this.drawingCanvas);
    Square mySquare = new Square(100);
    if (mySquare is IDraw)
    {
        IDraw drawSquare = mySquare;
        drawSquare.SetLocation((int)mouseLocation.X, (int)mouseLocation.Y);
        drawSquare.Draw(drawingCanvas);
    }
}
```

The `TappedRoutedEventArgs` parameter in this method provides useful information about the position of the mouse. In particular, the `GetPosition` method returns a `Point` structure that contains the x- and y-coordinates of the mouse. The code that you've added creates a new `Square` object. It then checks to verify that this object implements the `IDraw` interface. (This is good practice and helps to ensure that your code will not fail at runtime if you attempt to reference an object through an interface that it does not implement.) Next, it creates a reference to the object by using this interface. Remember that when you explicitly implement an interface, the methods defined by the interface are available only by creating a reference to that interface. (The `SetLocation` and `Draw` methods are private to the `Square` class and are available only through the `IDraw` interface.) The code then sets the location of the `Square` shape to the position of the user's finger or mouse. Note that the x- and y-coordinates in the `Point` structure are actually `double` values, so this code casts them to `ints`. Finally, the code calls the `Draw` method to display the `Square` object.

7. At the end of the `drawingCanvas_Tapped` method, add the following code shown in bold:

```
private void drawingCanvas_Tapped(object sender, TappedRoutedEventArgs e)
{
    ...
    if (mySquare is IColor)
    {
        IColor colorSquare = mySquare;
        colorSquare.SetColor(Colors.BlueViolet);
    }
}
```

This code tests the `Square` class to verify that it implements the `IColor` interface. If it does, the code creates a reference to the `Square` class through this interface and calls the `SetColor` method to set the color of the `Square` object to `Colors.BlueViolet`.



Important You must call `Draw` before you call `SetColor` because the `SetColor` method sets the color of the `Square` only if it has already been rendered. If you invoke `SetColor` before `Draw`, the color will not be set, and the `Square` object will not appear.

8. Return to the `DrawingPad.xaml` file in the Design View window, and then select the Canvas object.
9. In the list of events, locate the `RightTapped` event, and then double-click the **RightTapped** box.

This event occurs when the user taps, holds, and then releases from the canvas by using a finger or right-clicks on the canvas.

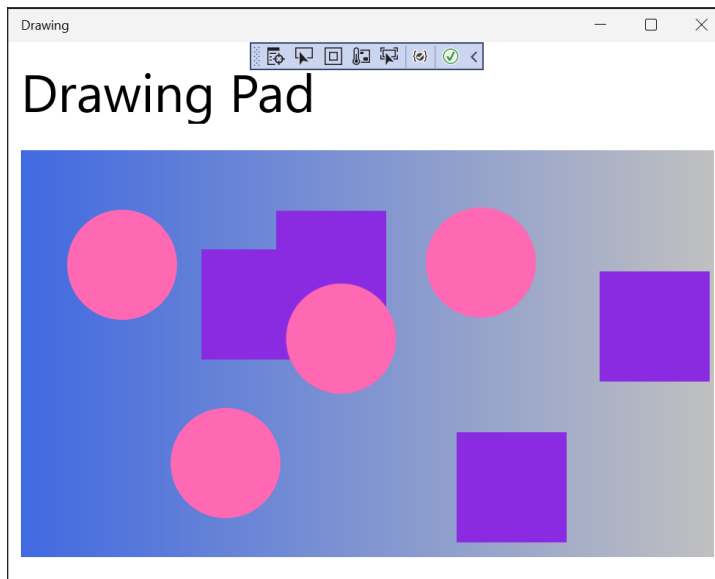
10. Add the following code shown in bold to the `drawingCanvas_RightTapped` method:

```
private void drawingCanvas_RightTapped(object sender, RightTappedRoutedEventArgs e)
{
    Point mouseLocation = e.GetPosition(this.drawingCanvas);
    Circle myCircle = new Circle(100);
    if (myCircle is IDraw)
    {
        IDraw drawCircle = myCircle;
        drawCircle.SetLocation((int)mouseLocation.X, (int)mouseLocation.Y);
        drawCircle.Draw(drawingCanvas);
    }

    if (myCircle is IColor)
    {
        IColor colorCircle = myCircle;
        colorCircle.SetColor(Colors.HotPink);
    }
}
```

The logic in this code is similar to the logic in the `drawingCanvas_Tapped` method, except that this code draws and fills a circle rather than a square.

11. On the **Debug** menu, select **Start Debugging** to build and run the application.
12. When the Drawing Pad window opens, tap or click anywhere on the canvas displayed in the window. A violet square should appear.
13. Tap, hold, and release, or right-click anywhere on the canvas. A pink circle should appear. You can click the left and right mouse buttons any number of times; each click will draw a square or circle at the mouse position.



14. Return to Visual Studio and stop debugging.

Abstract classes

You can implement the `ILandBound` and `IGrazable` interfaces discussed before the previous set of exercises in many different classes, depending on how many different types of mammals you want to model in your C# application. In situations such as this, it's typical for parts of the derived classes to share common implementations. For example, the duplication in the following two classes is obvious:

```
class Horse : Mammal, ILandBound, IGrazable
{
    ...
    void IGrazable.ChewGrass()
    {
        Console.WriteLine( "Chewing grass ");
        // code for chewing grass
    }
}
class Sheep : Mammal, ILandBound, IGrazable
{
    ...
    void IGrazable.ChewGrass()
    {
        Console.WriteLine( "Chewing grass ");
        // same code as horse for chewing grass
    }
}
```

Duplication in code is a warning sign. If possible, you should refactor code to avoid duplication and reduce any associated maintenance costs. One way to achieve this refactoring is to put the common implementation into a new class created specifically for this purpose. In effect, you can insert a new class into the class hierarchy, as shown in the following code example:

```
class GrazingMammal : Mammal, IGrazable
{
    ...
    void IGrazable.ChewGrass()
    {
        // common code for chewing grass
        Console.WriteLine( "Chewing grass ");
    }
}

class Horse : GrazingMammal, ILandBound
{
    ...
}

class Sheep : GrazingMammal, ILandBound
{
    ...
}
```

This is a good solution, but one thing is still not quite right: you can actually create instances of the `GrazingMammal` class (and the `Mammal` class, for that matter). This doesn't really make sense. The `GrazingMammal` class exists to provide a common default implementation. Its sole purpose is to be a

class from which to inherit. The `GrazingMammal` class is an abstraction of common functionality rather than an entity in its own right.

To declare that creating instances of a class is not allowed, you can declare that the class is abstract by using the `abstract` keyword, such as in the following example:

```
abstract class GrazingMammal : Mammal, IGrazable
{
    ...
}
```

If you now try to instantiate a `GrazingMammal` object, the code will not compile:

```
GrazingMammal myGrazingMammal = new GrazingMammal(...); // illegal
```

Abstract methods

An abstract class can contain abstract methods. An abstract method is similar in principle to a virtual method (covered in Chapter 12), except that it does not contain a method body. A derived class *must* override this method. An abstract method cannot be private. The following example defines the `DigestGrass` method in the `GrazingMammal` class as an abstract method; grazing mammals might use the same code for chewing grass, but they must provide their own implementation of the `DigestGrass` method. An abstract method is useful if it doesn't make sense to provide a default implementation in the abstract class but you want to ensure that an inheriting class provides its own implementation of that method.

```
abstract class GrazingMammal : Mammal, IGrazable
{
    public abstract void DigestGrass();
    ...
}
```



Note At first glance, an abstract class and an interface with default method implementations may appear to be similar, but they are fundamentally different beasts. An abstract class can contain fields, constructors, a destructor, private methods, and other members that you would typically find in any class. An interface is simply a specification of what a class should look like.

Sealed classes

Using inheritance is not always easy and requires forethought. If you create an interface or an abstract class, you are knowingly writing something that will be inherited from in the future. The trouble is that predicting the future is a difficult business. With practice and experience, you can develop the skills to craft a flexible, easy-to-use hierarchy of interfaces, abstract classes, and classes, but it takes effort as well as a solid understanding of the problem that you're modeling. To put it another way, unless you

consciously design a class with the intention of using it as a base class, it's extremely unlikely that it will function well as a base class.

With C#, you can use the `sealed` keyword to prevent a class from being used as a base class if you decide that it should not be. For example, in the following code, if any class attempts to use `Horse` as a base class, a compile-time error will be generated:

```
sealed class Horse : GrazingMammal, ILandBound
{
    ...
}
```



Note A sealed class cannot declare any virtual methods. An abstract class cannot be sealed.

Sealed methods

You can also use the `sealed` keyword to declare that an individual method in an unsealed class is sealed. This means that a derived class cannot override this method. You can seal only a method declared with the `override` keyword, and you declare the method as *sealed override*. You can think of the `interface`, `virtual`, `override`, and `sealed` keywords as follows:

- An interface introduces the name of a method.
- A virtual method is the first implementation of a method.
- An override method is another implementation of a method.
- A sealed method is the final implementation of a method.

Implementing and using an abstract class

The following exercises use an abstract class to rationalize some of the code that you developed in the previous exercise. The `Square` and `Circle` classes contain a high proportion of duplicate code. It makes sense to factor this code into an abstract class called `DrawingShape` because this will help to ease maintenance of the `Square` and `Circle` classes in the future.

To create the `DrawingShape` abstract class

1. Return to the Drawing project in Visual Studio.
2. In Solution Explorer, select the **Drawing** project in the Drawing solution. On the **Project** menu, select **Add Class**.

The Add New Item – Drawing dialog opens.

3. In the **Name** box, type **DrawingShape.cs**, and then select **Add**.

Visual Studio creates the class and displays it in the Code and Text Editor window.

4. In the `DrawingShape.cs` file, at the list at the top of the file, add the following `using` directives:

```
using Windows.UI;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Shapes;
using Windows.UI.Xaml.Controls;
```

The purpose of this class is to contain the code common to the `Circle` and `Square` classes. A program should not be able to instantiate a `DrawingShape` object directly.

5. Modify the definition of the `DrawingShape` class to declare it as abstract, as shown here in bold:

```
abstract class DrawingShape
{
}
```

6. Add the following protected variables shown in bold to the `DrawingShape` class:

```
abstract class DrawingShape
{
    protected int size;
    protected int locX = 0, locY = 0;
    protected Shape shape = null;
}
```

The `Square` and `Circle` classes both use the `locX` and `locY` fields to specify the location of the object on the canvas so that you can move these fields to the abstract class. Similarly, the `Square` and `Circle` classes both use a field to indicate the size of the object when it was rendered. Although it has a different name in each class (`sideLength` and `diameter`), semantically the field performs the same task in both classes. The name `size` is a good abstraction of the purpose of this field.

Internally, the `Square` class uses a `Rectangle` object to render itself on the canvas, and the `Circle` class uses an `Ellipse` object. Both of these classes are part of a hierarchy based on the abstract `Shape` class in .NET. The `DrawingShape` class uses a `Shape` field to represent both of these types.

7. Add the following constructor to the `DrawingShape` class:

```
abstract class DrawingShape
{
    ...
    public DrawingShape(int size)
    {
        this.size = size;
    }
}
```

This code initializes the `size` field in the `DrawingShape` object.

8. Add the `SetLocation` and `SetColor` methods to the `DrawingShape` class, as shown in bold in the code that follows. These methods provide implementations that are inherited by all classes that derive from the `DrawingShape` class. Notice that they are not marked as `virtual`, and a

derived class is not expected to override them. Also, the `DrawingShape` class is not declared as implementing the `IDraw` or `IColor` interfaces (interface implementation is a feature of the `Square` and `Circle` classes and not this abstract class), so these methods are simply declared as public.

```
abstract class DrawingShape
{
    ...
    public void SetLocation(int xCoord, int yCoord)
    {
        this.locX = xCoord;
        this.locY = yCoord;
    }

    public void SetColor(Color color)
    {
        if (this.shape is not null)
        {
            SolidColorBrush brush = new SolidColorBrush(color);
            this.shape.Fill = brush;
        }
    }
}
```

9. Add the `Draw` method to the `DrawingShape` class. Unlike the previous methods, this method is declared as `virtual`, and any derived classes are expected to override it to extend the functionality. The code in this method verifies that the `shape` field is not null and then draws it on the canvas. The classes that inherit this method must provide their own code to instantiate the shape object. (Remember that the `Square` class creates a `Rectangle` object and the `Circle` class creates an `Ellipse` object.)

```
abstract class DrawingShape
{
    ...
    public virtual void Draw(Canvas canvas)
    {
        if (this.shape is null)
        {
            throw new InvalidOperationException("Shape is null ");
        }

        this.shape.Height = this.size;
        this.shape.Width = this.size;
        Canvas.SetTop(this.shape, this.locY);
        Canvas.SetLeft(this.shape, this.locX);
        canvas.Children.Add(this.shape);
    }
}
```

You've now completed the `DrawingShape` abstract class. The next steps are to change the `Square` and `Circle` classes so that they inherit from this class and then to remove the duplicated code from the `Square` and `Circle` classes.

To modify the Square and Circle classes to inherit from the DrawingShape class

1. Display the code for the Square class in the Code and Text Editor window.
2. Modify the definition of the Square class so that it inherits from the DrawingShape class in addition to implementing the IDraw and IColor interfaces.

```
class Square : DrawingShape, IDraw, IColor
{
    ...
}
```

Notice that you must specify the class that the Square class inherits from before any interfaces that it implements.

3. In the Square class, remove the definitions of the `sideLength`, `rect`, `locX`, and `locY` fields. These fields are no longer necessary because they are now provided by the DrawingShape class.
4. Replace the existing constructor with the following code shown in bold, which calls the constructor in the base class:

```
class Square : DrawingShape, IDraw, IColor
{
    public Square(int sideLength)
        : base(sideLength)
    {
    }
    ...
}
```

Notice that the body of this constructor is empty because the base class constructor performs all the initialization required.

5. Remove the `IDraw.SetLocation` and `IColor.SetColor` methods from the Square class. The DrawingShape class provides the implementation of these methods.
6. Modify the definition of the Draw method. Declare it with `public override` and also remove the reference to the IDraw interface. Again, the DrawingShape class already provides the base functionality for this method, but you will extend it with the specific code required by the Square class.

```
public override void Draw(Canvas canvas)
{
    ...
}
```

7. Replace the body of the Draw method with the code shown here in bold:

```
public override void Draw(Canvas canvas)
{
    if (this.shape is not null)
    {
        canvas.Children.Remove(this.shape);
    }
    else
    {
        this.shape = new Rectangle();
    }

    base.Draw(canvas);
}
```

These statements instantiate the shape field inherited from the DrawingShape class as a new instance of the Rectangle class if it has not already been instantiated. They then call the Draw method in the DrawingShape class.

8. Repeat steps 2 through 7 for the Circle class, except that the constructor should be called Circle with a parameter called diameter, and in the Draw method, you should instantiate the shape field as a new Ellipse object. The complete code for the Circle class should look like this:

```
class Circle : DrawingShape, IDraw, IColor
{
    public Circle(int diameter)
        :base(diameter)
    {
    }

    public override void Draw(Canvas canvas)
    {
        if (this.shape is not null)
        {
            canvas.Children.Remove(this.shape);
        }
        else
        {
            this.shape = new Ellipse();
        }

        base.Draw(canvas);
    }
}
```

9. On the **Debug** menu, select **Start Debugging**.
10. When the Drawing Pad window opens, verify that Square objects appear when you left-click in the window, and Circle objects appear when you right-click in the window. The application should behave the same as before.
11. Return to Visual Studio and stop debugging.

Compatibility with the Windows Runtime revisited

Chapter 9, “Creating value types with enumerations and structures,” described how the Windows platform from Windows 8 onward implements the Windows Runtime (WinRT) as a layer on top of the native Windows APIs, providing a simplified programming interface for developers building unmanaged applications. (An unmanaged application is an application that does not run by using .NET; you build them by using a language such as C++ rather than C#.) Managed applications use the common language runtime (CLR) to run .NET applications.

.NET provides an extensive set of libraries and features. On Windows 7 and earlier versions, the CLR implements these features by using the native Windows APIs. If you’re building desktop or enterprise applications and services on Windows 11, this same feature set is still available (although .NET itself has been upgraded to version 6), and any C# applications that work on Windows 7 should run unchanged on Windows 11.

On Windows 11, UWP apps always run by using WinRT. This means that if you’re building UWP apps by using a managed language such as C#, the CLR actually invokes WinRT rather than the native Windows APIs. Microsoft has provided a mapping layer between the CLR and WinRT that can transparently translate requests to create .NET objects and invoke .NET methods into the equivalent object requests and method calls in WinRT. For example, when you create a .NET `Int32` value (an `int` in C#), this code is translated to create a value using the equivalent WinRT data type.

Although the CLR and WinRT have a large amount of overlapping functionality, not all the features of .NET have corresponding features in WinRT. Consequently, UWP apps have access to only a reduced subset of the types and methods provided by .NET. (IntelliSense in Visual Studio 2022 automatically shows the restricted view of available features when you use C# to build UWP apps, omitting the types and methods not available through WinRT.)

Conversely, WinRT provides a significant set of features and types that have no direct equivalent in .NET or that operate in a drastically different way to the corresponding features in .NET, and so cannot easily be translated. WinRT makes these features available to the CLR through a mapping layer that makes them look like .NET types and methods, and you can invoke them directly from managed code.

So, integration implemented by the CLR and WinRT enables the CLR to transparently use WinRT types, but it also supports interoperability in the reverse direction: you can define types by using managed code and make them available to unmanaged applications as long as these types conform to the expectations of WinRT. Chapter 9 highlights the requirements of structures in this respect. (Instance and static methods in structures are not available through WinRT, and private fields are unsupported.)

If you’re building classes with the intention that they be consumed by unmanaged applications through WinRT, your classes must follow these rules:

- Any public fields, and the parameters and return values of any public methods, must be WinRT types or .NET types that can be transparently translated by WinRT into WinRT types.

Examples of supported .NET types include conforming value types (such as structures and enumerations) and those corresponding to the C# primitives (`int`, `long`, `float`, `double`, `string`, and so on). Private fields are supported in classes, and they can be of any type available in .NET; they do not have to conform to WinRT.

- Classes cannot override methods of `System.Object` other than `ToString`, and they cannot declare protected constructors.
- The namespace in which a class is defined must be the same as the name of the assembly implementing the class. Additionally, the namespace name (and therefore the assembly name) must not begin with `Windows`.
- You cannot inherit from managed types in unmanaged applications through WinRT. Therefore, all public classes must be sealed. If you need to implement polymorphism, you can create a public interface and implement that interface on the classes that must be polymorphic.
- You can throw any exception type that is included in the subset of .NET available to UWP apps; however, you cannot create your own custom exception classes. If your code throws an unhandled exception when called from an unmanaged application, WinRT raises an equivalent exception in the unmanaged code.

WinRT has other requirements concerning features of C# code covered later in this book. These requirements will be highlighted as each feature is described.

Summary

In this chapter, you saw how to define and implement interfaces and abstract classes. The following table summarizes the various valid (yes) and invalid (no) keyword combinations when defining methods for interfaces, classes, and structs.

| Keyword | Interface | Abstract class | Class | Sealed class | Structure |
|------------------------|------------------|----------------|-------|--------------|-----------------|
| <code>abstract</code> | No | Yes | No | No | No |
| <code>new</code> | Yes ¹ | Yes | Yes | Yes | No ² |
| <code>override</code> | No | Yes | Yes | Yes | No ³ |
| <code>private</code> | No | Yes | Yes | Yes | Yes |
| <code>protected</code> | No | Yes | Yes | Yes | No ⁴ |
| <code>public</code> | No | Yes | Yes | Yes | Yes |
| <code>sealed</code> | No | Yes | Yes | Yes | No |
| <code>virtual</code> | No | Yes | Yes | No | No |

1 An interface can extend another interface and introduce a new method with the same signature.

2 Structures do not support inheritance, so they cannot hide methods.

3 Structures do not support inheritance, so they cannot override methods.

4 Structures do not support inheritance; a structure is implicitly sealed and cannot be derived from.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 14.
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

| To | Do this |
|---|--|
| Declare an interface | Use the interface keyword. For example:
<pre>interface IDemo { string GetName(); string GetDescription(); }</pre> |
| Implement an interface | Declare a class by using the same syntax as class inheritance, and then implement all the member functions of the interface. For example:
<pre>class Test : IDemo { public string IDemo.GetName() { ... } public string IDemo.GetDescription() { ... } }</pre> |
| Create an abstract class that can be used only as a base class, containing abstract methods | Declare the class by using the abstract keyword. For each abstract method, declare the method with the abstract keyword and without a method body. For example:
<pre>abstract class GrazingMammal { abstract void DigestGrass(); ... }</pre> |
| Create a sealed class that cannot be used as a base class | Declare the class by using the sealed keyword. For example:
<pre>sealed class Horse { ... }</pre> |

Using garbage collection and resource management

After completing this chapter, you will be able to:

- Explain the purpose of the garbage collector and how it works.
- Manage system resources by using garbage collection.
- Release a resource at a known point in time in an exception-safe manner by writing a `try/finally` statement.
- Release a resource at a known point in time in an exception-safe manner by writing a `using` statement.
- Implement the `IDisposable` and `IAsyncDisposable` interfaces to support exception-safe disposal in a class.

You've seen in earlier chapters how to create variables and objects, and you should understand how memory is allocated when you create variables and objects. (In case you don't remember, value types are created on the stack, and reference types are allocated memory from the heap.) Computers don't have infinite amounts of memory, so memory must be reclaimed when a variable or an object no longer needs it. Value types are destroyed and their memory reclaimed when they go out of scope. That's the easy bit. But what about reference types? You create an object by using the `new` keyword, but how and when is an object destroyed? That's what this chapter is all about.

The life and times of an object

First, let's recap what happens when you create an object. You create an object by using the `new` operator. The following example creates a new instance of the `Square` class discussed in Chapter 13, "Creating interfaces and defining abstract classes."

```
int sizeOfSquare = 99;  
Square mySquare = new Square(sizeOfSquare); // Square is a reference type
```


From your point of view, the new operation is a single step, but underneath, object creation is really a two-phase process:

1. The new operation allocates a chunk of raw memory from the heap. You have no control over this phase of an object's creation.
2. The new operation converts the chunk of raw memory to an object so the object can be initialized. You can control this phase by using a constructor.



Note If you're a C++ programmer, you should note that in C#, you cannot overload the new operation to control allocation.

After you create an object, you can access its members by using the dot operator (.). For example, the Square class includes a method named Draw that you can call:

```
mySquare.Draw();
```



Note This code is based on the version of the Square class that inherits from the DrawingShape abstract class and does not implement the IDraw interface explicitly. For more information, refer to Chapter 13.

When the mySquare variable goes out of scope, the Square object is no longer actively referenced. So, the object Square object can be destroyed, and the memory that it's using can be reclaimed. (This might not happen immediately, however, as you'll see later.)

Like object creation, object destruction is a two-phase process. The two phases of destruction exactly mirror the two phases of creation:

1. The .NET runtime performs some tidying up by releasing resources associated with the object. You can control this process by writing a *finalizer*.
2. The .NET runtime returns the memory previously belonging to the object back to the heap. The memory that the object lived in must be deallocated. You have no control over this phase.

The process of destroying an object and returning memory to the heap is known as *garbage collection*.



Note If you program in C++, keep in mind that C# does not have a delete operator. The .NET runtime controls when an object is destroyed.

Writing finalizers

You can use a finalizer to perform any tidying up that's required when an object is garbage-collected. A finalizer is a special method, a little like a constructor, except that the .NET runtime calls it after the final reference to an object has disappeared. The .NET runtime automatically clears up any managed

resources that an object uses, so in many of these cases, writing a finalizer is unnecessary. However, if a managed resource is large (such as a multidimensional array), it might make sense to make this resource available for immediate disposal by setting any references that the object has to this resource to `null`. Also, if an object references an unmanaged resource, either directly or indirectly, a finalizer can prove useful.



Note Indirect unmanaged resources are reasonably common. Examples include file streams, network connections, database connections, and other resources managed by Windows. If you open a file in a method, for example, you might want to add a finalizer that closes the file when the object is destroyed. However, there might be a better and timelier way to close the file depending on the structure of the code in your class. (See the discussion of the `using` statement later in this chapter for more information.)

The syntax for writing a finalizer is a tilde (~) followed by the name of the class. For example, here's a simple class that opens a file for reading in its constructor and closes the file in its finalizer. (Note that this is simply an example; I do not recommend that you always follow this pattern for opening and closing files.)

```
class FileProcessor
{
    FileStream file = null;

    public FileProcessor(string fileName)
    {
        this.file = File.OpenRead(fileName); // open file for reading
    }

    ~FileProcessor()
    {
        this.file.Close(); // close file
    }
}
```

There are some very important restrictions that apply to finalizers:

- Finalizers apply only to reference types. You cannot declare a finalizer in a value type, such as a `struct`:

```
struct MyStruct
{
    ~MyStruct() { ... } // compile-time error
}
```

- You cannot specify an access modifier (such as `public`) for a finalizer. You never call the finalizer in your own code; part of the .NET runtime, called the *garbage collector*, does this for you.

```
public ~FileProcessor() { ... } // compile-time error
```

- A finalizer cannot take any parameters. Again, this is because you never call the finalizer yourself.

```
~FileProcessor(int parameter) { ... } // compile-time error
```

- You cannot depend on when the finalizer runs, or even that it *will* run. When the application finishes, any objects that have not been finalized may simply be discarded and their finalizers not run. You can force the garbage collector to execute finalizers in your code, but this is absolutely not recommended. For this reason, finalization code should be concerned only with releasing resources. Never include any critical application logic in a finalizer.



Important C# finalizers are not like C++ destructors, and you shouldn't expect them to behave as such.

Internally, the C# compiler automatically translates a finalizer into an override of the `Object.Finalize` method. The compiler converts this finalizer:

```
class FileProcessor
{
    ~FileProcessor()
    {
        // your code goes here
    }
}
```

into this:

```
class FileProcessor
{
    protected override void Finalize()
    {
        try
        {
            // your code goes here
        }
        finally
        {
            base.Finalize();
        }
    }
}
```

The compiler-generated `Finalize` method contains the finalizer body within a `try` block, followed by a `finally` block that calls the `Finalize` method in the base class. (The `try` and `finally` keywords are described in Chapter 6, “Managing errors and exceptions.”) This ensures that a finalizer always calls its base-class finalizer, even if an exception occurs during the finalizer code.



Important Only the compiler can make this translation. You can't write your own method to override `Finalize`, and you can't call `Finalize` yourself.

Why use the garbage collector?

Before answering this question, consider that you can make more than one reference variable refer to the same object. In the following code example, the variables `myFp` and `referenceToMyFp` point to the same `FileProcessor` object:

```
FileProcessor myFp = new FileProcessor();  
FileProcessor referenceToMyFp = myFp;
```

How many references can you create to an object? As many as you want! There's just one problem: The .NET runtime must keep track of all these references. If the variable `myFp` disappears (by going out of scope), other variables (such as `referenceToMyFp`) might still exist, and the resources used by the `FileProcessor` object cannot be reclaimed (the file should not be closed). So, the lifetime of an object cannot be tied to a particular reference variable. An object can be destroyed and its memory made available for reuse only when *all* the references to it have disappeared.



Important You can never destroy an object yourself by using C# code. There just isn't any syntax to do it. Instead, the .NET runtime does it for you at a time of its own choosing.

Managing the lifetime of an object is complex, which is why the designers of C# decided to prevent your code from taking on this responsibility. If it were your responsibility to destroy objects, sooner or later one of the following situations would arise:

- You'd forget to destroy the object. This would mean that the object's finalizer (if it had one) would not be run, tidying up would not occur, and memory would not be returned to the heap. You could quite easily run out of memory.
- You'd try to destroy an active object to which one or more variables hold a reference—known as a *dangling reference*. A dangling reference refers either to unused memory or to a completely different object that now occupies the same piece of memory. Either way, the outcome of using a dangling reference would be undefined at best and a security risk at worst. All bets would be off.
- You'd try to destroy the same object more than once. This might or might not be disastrous, depending on the code in the finalizer.

These problems are unacceptable in a language like C#, which places robustness and security high on its list of design goals. So, the garbage collector destroys objects for you. The garbage collector makes the following guarantees:

- Every object will be destroyed. When a program ends, all outstanding objects will be destroyed. However, any finalizers that haven't been run by the time the program finishes might not be run at all.
- Every object will be destroyed at most once.
- Every object will be destroyed only when it becomes unreachable—that is, when there are no references to the object in the process running your application.

These guarantees are tremendously useful, and they free you, the programmer, from tedious house-keeping chores that are easy to get wrong. They afford you the luxury to concentrate on the logic of the program itself and be more productive.

When does garbage collection occur? This might seem like a strange question. After all, surely garbage collection occurs when an object is no longer needed. Well, it does—but not necessarily immediately. Garbage collection can be an expensive process, so the .NET runtime collects garbage only when it needs to—for example, when available memory is running low or the size of the heap has exceeded the system-defined threshold—and then it collects as much as it can. Performing a few large sweeps of memory is more efficient than performing lots of little dustings.



Note You can invoke the garbage collector in a program by using the `GC` static class in the `System` namespace. This class implements several static methods with which you can control some of the actions the garbage collector performs.

You can trigger garbage collection by calling the static method `Collect` of the `GC` class. However, except in a few cases, this is not recommended. The `GC.Collect` method starts the garbage collector, but the process runs asynchronously—that is, the `GC.Collect` method does not wait for garbage collection to be complete before it returns, so you still don't know whether your objects have been destroyed.

You can also force an app to wait for all objects' finalizers to be run using the `GC.WaitForPendingFinalizers` method. Again, this is not recommended as it encourages developers to introduce dependencies in the finalization process. In general, let the .NET runtime decide when it's best to collect garbage.

One feature of the garbage collector is that you don't know—and should not rely on—the order in which objects will be destroyed. The final point to understand is arguably the most important: finalizers do not run until objects are garbage-collected and, as stated, might not run at all. Consequently, you should never write code that depends on finalizers running in a particular sequence or at a specific point in your application.

How does the garbage collector work?

The garbage collector runs in its own thread and can execute only at certain times—typically when your application reaches the end of a method. While it runs, other threads running in your application will temporarily halt because the garbage collector might need to move objects around and update object references, and it cannot do this while objects are in use.



Note A *thread* is a separate path of execution in an application. Windows uses threads to enable an application to perform multiple operations concurrently.

The garbage collector is a complex piece of software that is self-tuning and implements some optimizations to try to balance the need to keep memory available with the requirement to maintain the performance of the application. The details of the internal algorithms and structures used by the garbage collector are beyond the scope of this book (and Microsoft continually refines the way in which the garbage collector performs its work), but at a high level, the steps that the garbage collector takes are as follows:

1. It builds a map of all reachable objects. It does this by repeatedly following reference fields inside objects. The garbage collector builds this map very carefully and ensures that circular references do not cause infinite recursion. Any object *not* in this map is deemed to be unreachable.
2. It checks whether any of the unreachable objects has a finalizer that needs to be run (a process called *finalization*). Any unreachable object that requires finalization is placed in a special queue called the *freachable queue* (pronounced F-reachable).
3. It deallocates the remaining unreachable objects (those that don't require finalization) by moving the reachable objects down the heap, thus defragmenting the heap and freeing memory at its top. When the garbage collector moves a reachable object, it also updates any references to the object.
4. It allows all other threads to resume and then finalizes the unreachable objects that require finalization (now in the *freachable queue*) by running the `Finalize` methods on its own thread.

Recommendations

Writing classes that contain finalizers adds complexity to your code and the garbage collection process, which makes your program run more slowly. If your program doesn't contain any finalizers, the garbage collector doesn't need to place unreachable objects in the *freachable queue* and finalize them. Clearly, not doing something is faster than doing something. Therefore, avoid using finalizers except when you really need them to reclaim unmanaged resources. (You can consider a `using` statement instead, as will be described later in this chapter.)

Be very careful when you write a finalizer. In particular, be aware that if your finalizer calls other objects, those other objects might have already had *their* finalizer called by the garbage collector. Remember, the order of finalization is not guaranteed. Therefore, ensure that finalizers do not depend on or overlap each other. For example, don't have two finalizers that try to release the same resource.

Resource management

Sometimes it's inadvisable to release a resource in a finalizer. Some resources are just too valuable to lie around waiting for an arbitrary length of time until the garbage collector releases them.

Scarce resources such as memory, database connections, and file handles need to be released, and they need to be released as soon as possible. In these situations, your only option is to release the resource yourself. You can achieve this by creating a *disposal method*—a method that explicitly disposes of a resource. If a class has a disposal method, you can call it and control when the resource is released.



Note The term *disposal method* refers to the purpose of the method rather than its name. A disposal method can be named using any valid C# identifier.

Disposal methods

An example of a class that implements a disposal method is the `TextReader` class from the `System.IO` namespace. This class provides a mechanism to read characters from a sequential stream of input. The `TextReader` class contains a virtual method named `Close`, which closes the stream. The `StreamReader` class (which reads characters from a stream, such as an open file) and the `StringReader` class (which reads characters from a string) both derive from `TextReader`, and both override the `Close` method.

Here's an example that reads lines of text from a file by using the `StreamReader` class and then displays them on the screen:

```
TextReader reader = new StreamReader(filename);
string line;
while ((line = reader.ReadLine()) is not null)
{
    Console.WriteLine(line);
}
reader.Close();
```

The `ReadLine` method reads the next line of text from the stream into a string. The `ReadLine` method returns `null` if there's nothing left in the stream. It's important to call `Close` when you've finished with `reader` to release the file handle and associated resources. However, there's a problem with this example: it's not safe from exceptions. If the call to `ReadLine` or `WriteLine` throws an exception, the call to `Close` will not happen; it will be bypassed. If this happens often enough, you'll run out of file handles and be unable to open any more files.

Exception-safe disposal

One way to ensure that a disposal method (such as `Close`) is always called, regardless of whether there is an exception, is to call the disposal method within a `finally` block. Here's the preceding example, adjusted to use this technique:

```
TextReader reader = new StreamReader(filename);
try
{
    string line;
    while ((line = reader.ReadLine()) is not null)
    {
        Console.WriteLine(line);
    }
}
finally
{
    reader.Close();
}
```

Using a `finally` block like this works, but it has several drawbacks that make it a less-than-ideal solution:

- It quickly becomes unwieldy if you have to dispose of more than one resource. (You end up with nested `try` and `finally` blocks.)
- In some cases, you might need to modify the code to make it fit this idiom. For example, you might need to reorder the declaration of the resource reference, remember to initialize the reference to `null`, and remember to check that the reference isn't `null` in the `finally` block.
- It fails to create an abstraction of the solution. This means the solution is hard to understand, and you must repeat the code everywhere you need this functionality.
- The reference to the resource remains in scope after the `finally` block. This means you can accidentally try to use the resource after it has been released.

The `using` statement is designed to solve all these problems.

The `using` statement and the `IDisposable` interface

The `using` statement provides a clean mechanism for controlling the lifetimes of resources. You can create an object, and this object will be destroyed when the `using` statement block finishes.



Important Do not confuse the `using` statement shown in this section with the `using` directive that brings a namespace into scope. It's unfortunate that the same keyword has two different meanings.

The syntax for a `using` statement is as follows:

```
using ( type variable = initialization )
{
    StatementBlock
}
```

Here's the best way to ensure that your code always calls `Close` on a `TextReader`:

```
using (TextReader reader = new StreamReader(filename))
{
    string line;
    while ((line = reader.ReadLine()) is not null)
    {
        Console.WriteLine(line);
    }
}
```

This `using` statement is equivalent to the following transformation:

```
{
    TextReader reader = new StreamReader(filename);
    try
    {
```



```

    string line;
    while ((line = reader.ReadLine()) is not null)
    {
        Console.WriteLine(line);
    }
}
finally
{
    if (reader is not null)
    {
        ((IDisposable)reader).Dispose();
    }
}
}

```



Note The `using` statement introduces its own block for scoping purposes. This means the variable you declare in a `using` statement automatically goes out of scope at the end of the embedded statement, and you cannot accidentally attempt to access a disposed resource.

The variable you declare in a `using` statement must be of a type that implements the `IDisposable` interface. The `IDisposable` interface lives in the `System` namespace and contains just one method, named `Dispose`:

```

namespace System
{
    interface IDisposable
    {
        void Dispose();
    }
}

```

The purpose of the `Dispose` method is to free the resources used by an object. It just so happens that the `StreamReader` class implements the `IDisposable` interface, and its `Dispose` method calls `Close` to close the stream. You can employ a `using` statement as a clean, exception-safe, and robust way to ensure that a resource is always released. This approach solves all the problems that existed in the manual `try/finally` solution. You now have a solution that does the following:

- Scales well if you need to dispose of multiple resources.
- Doesn't distort the logic of the program code.
- Abstracts away the problem and avoids repetition.
- Achieves robustness. You can't accidentally reference the variable declared within the `using` statement (in this case, `reader`) after the `using` statement has ended because it's not in scope anymore; you'll get a compile-time error.

Calling the Dispose method from a finalizer

When writing your own classes, should you write a finalizer? Or should you implement the `IDisposable` interface so that instances of your class can be managed by a `using` statement?

A call to a finalizer might or might not happen, and when it does happen, you don't know when it will run. On the other hand, you know exactly when a call to the `Dispose` method occurs. However, you can't be sure that it will actually happen because it relies on the programmer who's using your classes to remember to write a `using` statement or call the `Dispose` method directly.

However, it is possible to ensure that the `Dispose` method always runs by calling it from the finalizer. This acts as a useful backup. You might forget to call the `Dispose` method, but at least you can be sure that it will be called—even if it's only when the program shuts down.

You'll investigate this feature in detail in the exercises at the end of the chapter, but here's an example of how you might implement the `IDisposable` interface:

```
class Example : IDisposable
{
    private Resource scarce;        // scarce resource to manage and dispose
    private bool disposed = false; // flag to indicate whether the resource
                                    // has already been disposed

    ...
    ~Example()
    {
        this.Dispose(false);
    }

    public virtual void Dispose()
    {
        this.Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!this.disposed)
        {
            if (disposing)
            {
                // release large, managed resource here
                ...
            }
            // release unmanaged resources here
            ...
            this.disposed = true;
        }
    }
}
```

```

public void SomeBehavior() // example method
{
    checkIfDisposed();
    ...
}

...

private void checkIfDisposed()
{
    if (this.disposed)
    {
        throw new ObjectDisposedException("Example: object has been disposed");
    }
}
}

```

Notice the following features of the `Example` class:

- The class implements the `IDisposable` interface.
- The public `Dispose` method can be called at any time by your application code.
- The public `Dispose` method calls a protected and overloaded version of the `Dispose` method that takes a Boolean parameter (described in an upcoming bullet point), passing the value `true` as the argument. This method actually performs the resource disposal.
- The finalizer calls the protected and overloaded version of the `Dispose` method that takes a Boolean parameter, passing the value `false` as the argument. The finalizer is called only by the garbage collector when your object is being finalized.
- You can safely call the protected `Dispose` method multiple times. The `disposed` variable indicates whether the method has already been run and is a safety feature to prevent the method from attempting to dispose of the resources multiple times if it's called concurrently. (Your application might call `Dispose`, but before the method completes, your object might be subject to garbage collection and the `Dispose` method run again by the .NET runtime from the finalizer.) The resources are released only the first time the method runs.
- The protected `Dispose` method supports disposal of managed resources (such as a large array) and unmanaged resources (such as a file handle). If the disposing parameter is `true`, this method must have been called from the public `Dispose` method. In this case, the managed and unmanaged resources are all released. If the disposing parameter is `false`, this method must have been called from the finalizer, and the garbage collector is finalizing the object. In this case, it's not necessary (or exception-safe) to release the managed resources because they will be, or might already have been, handled by the garbage collector, so only the unmanaged resources are released.
- The public `Dispose` method calls the static `GC.SuppressFinalize` method. This method stops the garbage collector from calling the finalizer on this object because the object has already been finalized.
- All the regular methods of the class (such as `SomeBehavior`) check to see whether the object has already been discarded. If it has, they throw an exception.

Implementing exception-safe disposal

In the following exercises, you'll examine how the `using` statement helps to ensure that resources used by objects in your applications can be released promptly, even if an exception occurs in your application code. First, you'll implement a simple class that implements a finalizer and examine when this finalizer is invoked by the garbage collector.



Note The `Calculator` class created in these exercises is intended only to illustrate the essential principles of garbage collection. The class does not actually consume any significant managed or unmanaged resources. You would not normally create a finalizer or implement the `IDisposable` interface for such a simple class as this.

To create a simple class that uses a finalizer

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **GarbageCollection** solution, which is located in the `\Microsoft Press\VCSBS\Chapter 14\GarbageCollection` folder in your **Documents** folder.
3. Open the `Program.cs` file. This file contains the same test harness that you've seen in previous chapters.
4. On the **Project** menu, select **Add Class**.
5. In the Add New Item – GarbageCollection dialog, ensure that the **Class** template is selected. In the **Name** box, type **Calculator.cs**, and then select **Add**.
6. Add the following public `Divide` method (shown in bold) to the `Calculator` class:

```
class Calculator
{
    public int Divide(int first, int second)
    {
        return first / second;
    }
}
```

This is a very straightforward method that divides the first parameter by the second and returns the result. It is provided just to add a bit of functionality that can be called by an application.

7. Before the `Divide` method, add the public constructor shown in bold in the following code:

```
class Calculator
{
    public Calculator()
    {
        Console.WriteLine("Calculator being created");
    }
    ...
}
```

The purpose of this constructor is to enable you to verify that a `Calculator` object has been successfully created.

8. Add the finalizer shown in bold in the following code, after the constructor:

```
class Calculator
{
    ...
    ~Calculator()
    {
        Console.WriteLine("Calculator being finalized");
    }
    ...
}
```

This finalizer simply displays a message so you can see when the garbage collector runs and finalizes instances of this class. When writing classes for real-world applications, you would not normally output text in a finalizer.

9. Display the `Program.cs` file in the Code and Text Editor window.
10. In the `doWork` method, replace the `// TODO: comment` with the following statements shown in bold:

```
static void doWork()
{
    var calculator = new Calculator();
    Console.WriteLine($"120 / 15 = {calculator.Divide(120, 15)}");
    calculator = null;
    Console.WriteLine("Program finishing");
}
```

This code creates a `Calculator` object, calls the `Divide` method of this object and displays the result, and then outputs a message as the method and program finishes.

11. On the **Debug** menu, select **Start Without Debugging**. Verify that the program displays the following series of messages:

```
Calculator being created
120 / 15 = 8
Program finishing
```

Notice that the finalizer for the `Calculator` object doesn't run—the message `Calculator being finalized` never appears.

12. In the console window, press **Enter** and return to Visual Studio 2022.

The .NET runtime guarantees that all objects created by your applications will be subject to garbage collection, but you cannot always be sure that any finalizers will run. In the exercise, the program was very short-lived, and when the .NET runtime tidied up as the program finished, there was no need to run the finalizer.

- 13.** In the Program.cs file, add the statements shown in bold to the end of the Main function:

```
static void Main(string[] args)
{
    try
    {
        doWork();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }

    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

These two statements force the garbage collector to run and wait for all finalizers to complete.

- 14.** On the **Debug** menu, select **Start Without Debugging**. Verify that the program displays the following series of messages:

```
Calculator being created
120 / 15 = 8
Program finishing
Calculator being finalized
```

In a more substantial and long-running application, you'll likely see garbage collection occur periodically, and the finalizers will be run automatically. However, the point of this exercise is that not all objects will necessarily be finalized before an application terminates.

In applications with classes that consume scarce resources, the objects that your applications create might retain their resources until the application finishes unless you provide a means of disposal. If the resource is a file, this could prevent other users from being able to access that file; if the resource is a database connection, your application could prevent other users from being able to connect to the same database. Ideally, you want to free resources as soon as you have finished using them rather than wait for the application to terminate. You require a more deterministic approach to resource management.

In the next exercise, you will implement the `IDisposable` interface in the `Calculator` class and enable the program to finalize `Calculator` objects at a time of its choosing.

To implement the `IDisposable` interface

1. Display the Calculator.cs file in the Code and Text Editor window.
2. Modify the definition of the `Calculator` class so that it implements the `IDisposable` interface, as shown here in bold:

```
class Calculator : IDisposable
{
    ...
}
```

3. Add the following method shown in bold, named `Dispose`, to the end of the `Calculator` class. This method is required by the `IDisposable` interface:

```
class Calculator : IDisposable
{
    ...
    public void Dispose()
    {
        Console.WriteLine("Calculator being disposed");
    }
}
```

You would normally add code to the `Dispose` method to release the resources held by the object. There are none in this case, and the purpose of the `Console.WriteLine` statement in this method is just to let you see when the `Dispose` method is run. However, in a real-world application, there would likely be some duplication of code between the finalizer and the `Dispose` method. To remove this duplication, you would typically place this code in one place and call it from the other. But because you cannot explicitly invoke a finalizer from the `Dispose` method, it makes sense instead to call the `Dispose` method from the finalizer and place the logic that releases resources in the `Dispose` method.

4. Modify the finalizer so that it calls the `Dispose` method, as shown in bold in the following code. (Leave the statement displaying the message in place in the finalizer so that you can see when it's being run by the garbage collector.)

```
~Calculator()
{
    Console.WriteLine("Calculator being finalized");
    this.Dispose();
}
```

When you want to destroy a `Calculator` object in an application, the `Dispose` method does not run automatically. Your code must either call it explicitly (with a statement such as `calculator.Dispose()`) or create the `Calculator` object within a `using` statement. In your program, you'll adopt the latter approach.

5. Display the `Program.cs` file in the Code and Text Editor window. Modify the statements in the `doWork` method that create the `Calculator` object and call the `Divide` method, as shown here in bold:

```
static void doWork()
{
    using (Calculator calculator = new Calculator())
    {
        Console.WriteLine($"120 / 15 = {calculator.Divide(120, 15)}");
    }

    Console.WriteLine("Program finishing");
}
```

6. In the `Main` method, comment out the statements that invoke the garbage collector and wait for finalizers to complete:

```
static void Main(String[] args)
{
    ...
    // GC.Collect();
    // GC.WaitForPendingFinalizers();
}
```

7. On the **Debug** menu, select **Start Without Debugging**. Verify that the program now displays the following series of messages:

```
Calculator being created
120 / 15 = 8
Calculator being disposed
Program finishing
```

The `using` statement causes the `Dispose` method to run before the statement that displays the `Program finishing` message.

8. In the console window, press **Enter** and return to Visual Studio 2022.
9. In the `Main` method, uncomment the two calls to the garbage collector to reinstate them. This will simulate the sequence of events that can occur in a longer-lived application that happens to run the garbage collector at some point.

10. On the **Debug** menu, select **Start Without Debugging**. You should now see the following set of messages:

```
Calculator being created
120 / 15 = 8
Calculator being disposed
Program finishing
Calculator being finalized
Calculator being disposed
```

You can see that when the finalizer for the `Calculator` object runs, it calls the `Dispose` method again. This is clearly a waste of processing. Disposing of the resources held by an object more than once might or might not be disastrous, but it's definitely not good practice. The recommended approach to resolving this problem is to add a private `Boolean` field to the class to indicate whether the `Dispose` method has already been invoked and then examine this field in the `Dispose` method.

11. In the console window, press **Enter** and return to Visual Studio 2022.

To prevent an object from being disposed of more than once

1. Display the Calculator.cs file in the Code and Text Editor window.
2. Add a private Boolean field called `disposed` to the start of the `Calculator` class. Initialize the value of this field to `false`, as shown in bold in the following code:

```
class Calculator : IDisposable
{
    private bool disposed = false;
    ...
}
```

The purpose of this field is to track the state of this object and indicate whether the `Dispose` method has been invoked.

3. Modify the code in the `Dispose` method to display the message only if the `disposed` field is false. After displaying the message, set the `disposed` field to true, as shown here in bold:

```
public void Dispose()
{
    if (!this.disposed)
    {
        Console.WriteLine("Calculator being disposed");
    }

    this.disposed = true;
}
```

4. On the **Debug** menu, select **Start Without Debugging**. Notice that the program displays the following series of messages:

```
Calculator being created
120 / 15 = 8
Calculator being disposed
Program finishing
Calculator being finalized
```

The `Calculator` object is now discarded only once, but the finalizer is still running. Again, this is a waste; there's little point in running a finalizer for an object that's already released its resources.

5. In the console window, press **Enter** and return to Visual Studio 2022.
6. In the `Calculator` class, add the following statement shown in bold to the end of the `Dispose` method:

```
public void Dispose()
{
    if (!this.disposed)
    {
        Console.WriteLine("Calculator being disposed");
    }

    this.disposed = true;
    GC.SuppressFinalize(this);
}
```

Using the `SuppressFinalize` method of the `GC` class, you can indicate that the garbage collector should not perform finalization on the specified object.



Important The `GC` class exposes several methods with which you can configure the garbage collector. However, as stated, it's usually better to let the .NET runtime manage the garbage collector itself because you can seriously impair the performance of your application if you call these methods in judiciously.

7. On the **Debug** menu, select **Start Without Debugging**. Notice that the program displays the following series of messages:

```
Calculator being created
120 / 15 = 8
Calculator being disposed
Program finishing
```

You can see that the finalizer is no longer running because the `Calculator` object has already been disposed of before the program finishes.

8. In the console window, press **Enter** and return to Visual Studio 2022.

Thread safety and the `Dispose` method

The example of using the `disposed` field to prevent an object from being discarded multiple times works well in most cases, but keep in mind that you have no control over when the finalizer runs. In the exercises in this chapter, it always executes as the program finishes, but this might not always be the case; it can run anytime after the last reference to an object has disappeared. So, it's possible that the finalizer could actually be invoked by the garbage collector on its own thread while the `Dispose` method is being run, especially if the `Dispose` method has to do a significant amount of work.

You could reduce the possibility of resources being released multiple times by moving the statement that sets the `disposed` field to `true` closer to the start of the `Dispose` method, but in this case you run the risk of not freeing the resources at all if an exception occurs after you've set this variable but before you've released them.

To eliminate the chances of two concurrent threads disposing of the same resources in the same object simultaneously, you can write your code in a thread-safe manner by embedding it in a `C#` `lock` statement, like this:

```
public void Dispose()
{
    lock(this)
    {
        if (!disposed)
        {
            Console.WriteLine("Calculator being disposed");
        }
    }
}
```

```

        this.disposed = true;
        GC.SuppressFinalize(this);
    }
}

```

The purpose of the `lock` statement is to prevent the same block of code from being run at the same time on different threads. The argument to the `lock` statement (`this` in the preceding example) should be a reference to an object. The code between the curly braces defines the scope of the `lock` statement. When execution reaches the `lock` statement, if the specified object is currently locked, the thread requesting the lock is blocked, and the code is suspended at this point. When the thread that currently holds the lock reaches the closing curly brace of the `lock` statement, the lock is released, enabling the blocked thread to acquire the lock itself and continue. However, by the time this happens, the `disposed` field will have been set to `true`, so the second thread will not attempt to perform the code in the `if (!disposed)` block.

Using locks in this manner is safe but can impair performance. An alternative approach is to use the strategy described earlier in this chapter, whereby only the repeated disposal of managed resources is suppressed. (It is not exception-safe to dispose of managed resources more than once; you will not compromise the security of your computer, but you might affect the logical integrity of your application if you attempt to dispose of a managed object that no longer exists.) This strategy implements overloaded versions of the `Dispose` method; the `using` statement calls `Dispose()`, which in turn runs the statement `Dispose(true)`, while the finalizer invokes `Dispose(false)`. Managed resources are freed only if the parameter to the overloaded version of the `Dispose` method is `true`. For more information, refer to the example in the section “Calling the `Dispose` method from a finalizer.”

The purpose of the `using` statement is to ensure that an object is always discarded, even if an exception occurs while it’s being used. In the final exercise in this chapter, you’ll verify that this is the case by generating an exception in the middle of a `using` block.

To verify that an object is disposed of after an exception

1. Display the `Program.cs` file in the Code and Text Editor window.
2. Modify the statement that calls the `Divide` method of the `Calculator` object as shown in bold:

```

static void doWork()
{
    using (Calculator calculator = new Calculator())
    {
        Console.WriteLine($"120 / 0 = {calculator.Divide(120, 0)}");
    }
    Console.WriteLine("Program finishing");
}

```

The amended statement attempts to divide 120 by 0.

3. In the `Main` method, change the catch statement to catch the `InvalidOperationException` exception:

```
static void Main(String[] args)
{
    try
    {
        doWork();
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine(ex.Message);
    }

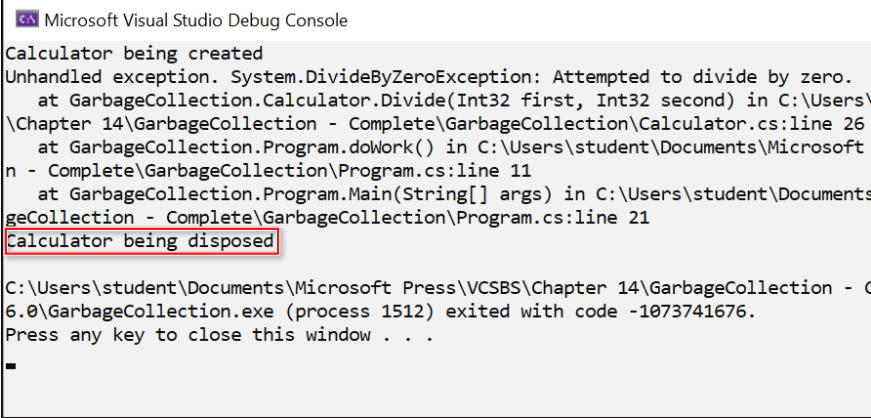
    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

This modification allows the `DivideByZeroException` exception thrown by the `doWork` method to remain unhandled, so you can see the effect this situation has on the garbage collector.

4. On the **Debug** menu, select **Start Without Debugging**.

As you might have anticipated, the application throws a `DivideByZeroException` exception.

5. Verify that the message `Calculator being disposed` appears after the unhandled exception in the console window when the program terminates. Garbage collection has still occurred.



```
Microsoft Visual Studio Debug Console
Calculator being created
Unhandled exception. System.DivideByZeroException: Attempted to divide by zero.
   at GarbageCollection.Calculator.Divide(Int32 first, Int32 second) in C:\Users\
\Chapter 14\GarbageCollection - Complete\GarbageCollection\Calculator.cs:line 26
   at GarbageCollection.Program.doWork() in C:\Users\student\Documents\Microsoft
n - Complete\GarbageCollection\Program.cs:line 11
   at GarbageCollection.Program.Main(String[] args) in C:\Users\student\Documents
geCollection - Complete\GarbageCollection\Program.cs:line 21
Calculator being disposed
C:\Users\student\Documents\Microsoft Press\VCBS\Chapter 14\GarbageCollection - C
6.0\GarbageCollection.exe (process 1512) exited with code -1073741676.
Press any key to close this window . . .
```

6. In the console window, press **Enter** and return to Visual Studio 2022.

Handling asynchronous disposal

The `Dispose` method of the `IDisposable` interface is synchronous. If resource disposal takes a significant amount of time, the thread calling the `Dispose` method will be blocked until disposal is complete. If this is the thread that handles the user interface, then your application can become unresponsive and frustrating to use. To combat this, you can implement asynchronous disposal.

To indicate that an object should be disposed asynchronously, you use the `await` operator with the `using` statement that creates the resource, like this:

```
await using (BigDisposableType myObject = new BigDisposableType())
{
    ...
}
```

While the object is being disposed, at the end of the `using` block, the current thread will be freed to perform other work. When the disposal is complete, the thread can switch back to the first statement after the closing brace.



Note You'll learn more about the `await` operator and asynchronous operations, and ensuring that apps remain responsive, in Chapter 24, "Improving response time by performing asynchronous operations."

To support asynchronous disposal, the class must implement the `IAsyncDisposable` interface. This interface exposes a single method: `DisposeAsync`. This method has a similar purpose to the `Dispose` method in the `IDisposable` interface; you use it to release resources created by the object. A common implementation pattern for the `DisposeAsync` method is to delegate asynchronous disposal of large resources to another method, typically named `DisposeAsyncCore`, and then use the `Dispose` method of the `IDisposable` interface to release any other objects:

```
class BigDisposableType: IDisposable, IAsyncDisposable
{
    private bool disposed = false;
    ...

    public void Dispose()
    {
        if (!this.disposed)
        {
            // Perform synchronous disposal logic
            ...
        }

        this.disposed = true;
        GC.SuppressFinalize(this);
    }

    public async ValueTask DisposeAsync()
    {
        await this.DisposeAsyncCore();
        this.Dispose();
    }

    private Task DisposeAsyncCore()
    {
        // Dispose of large objects asynchronously
        ...
    }
}
```

Chapter 24 covers the ins and outs of asynchronous operations in much more detail.

Summary

In this chapter, you saw how the garbage collector works and how .NET uses it to dispose of objects and reclaim memory. You learned how to write a finalizer to clean up the resources used by an object when memory is recycled by the garbage collector. You also saw how to use the `using` statement to implement exception-safe disposal of resources and how to implement the `IDisposable` interface to support this form of object disposal.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 15, “Implementing properties to access fields.”
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

| To | Do this |
|--|--|
| Write a finalizer | Write a method whose name is the same as the name of the class and is prefixed with a tilde (~). The method must not have an access modifier (such as <code>public</code>) and cannot have any parameters or return a value. For example:
<pre>class Example { ~Example() { ... } }</pre> |
| Call a finalizer | You can't call a finalizer. Only the garbage collector can call a finalizer. |
| Force garbage collection (not recommended) | Call <code>GC.Collect</code> . |
| Release a resource at a known point in time (but at the risk of resource leaks if an exception interrupts the execution) | Write a disposal method (a method that disposes of a resource) and call it explicitly from the program. For example:
<pre>class TextReader { ... public virtual void Close() { ... } } class Example { void Use() { TextReader reader = ...; // use reader reader.Close(); } }</pre> |

| To | Do this |
|--|--|
| Support exception-safe disposal in a class | Implement the <code>IDisposable</code> interface. For example: <pre>class SafeResource : IDisposable { ... public void Dispose() { // Dispose resources here } }</pre> |
| Implement exception-safe disposal for an object that implements the <code>IDisposable</code> interface | Create the object in a <code>using</code> statement. For example: <pre>using (SafeResource resource = new SafeResource()) { // Use SafeResource here ... }</pre> |

PART III

Understanding the C# object model

| | | |
|-------------------|---|-----|
| CHAPTER 15 | Implementing properties to access fields | 365 |
| CHAPTER 16 | Handling binary data and using indexers | 395 |
| CHAPTER 17 | Introducing generics | 413 |
| CHAPTER 18 | Using collections | 445 |
| CHAPTER 19 | Enumerating collections | 469 |
| CHAPTER 20 | Decoupling application logic and
handling events | 483 |
| CHAPTER 21 | Querying in-memory data by using query
expressions | 513 |
| CHAPTER 22 | Operator overloading | 537 |

Parts I and II introduced you to the core syntax of the C# language and showed you how to build new types by using structures, enumerations, and classes. You also saw how the common language runtime (CLR) manages the memory used by variables and objects when a program runs, and you should now understand the lifecycle of C# objects. The chapters in Part III build on this information, showing you how to use C# to create extensible components—highly functional data types that you can reuse in many applications.

In Part III, you'll learn about more advanced features of C#, such as properties, indexers, generics, and collection classes. You'll see how you can build responsive systems by using events and how you can use delegates to invoke the application logic of one class from another without closely coupling the classes—a powerful technique that enables you to construct highly extensible systems. You'll also learn about Language-Integrated Query (LINQ), which enables you to perform complex queries over collections of objects in a clear and natural manner. And you'll see how to overload operators to customize the way in which common C# operators function over your own classes and structures.

Implementing properties to access fields

After completing this chapter, you will be able to:

- Encapsulate logical fields by using properties.
- Control read and write access to properties by declaring get accessors and set accessors.
- Explain the restrictions of properties.
- Create interfaces that declare properties.
- Implement interfaces containing properties by using structures and classes.
- Generate properties automatically based on field definitions.
- Use properties to initialize objects.
- Implement records based on property values.

This chapter looks at how to define and use properties to encapsulate fields and data in a class. Previous chapters emphasized that you should make the fields in a class private and provide methods to store values in them and to retrieve their values. This approach ensures safe and controlled access to fields.

You can also use this approach to encapsulate additional logic and rules concerning the permitted values. However, the syntax for accessing a field in this way is unnatural. When you want to read or write a variable, you normally use an assignment statement, so calling a method to achieve the same effect on a field (which is, after all, just a variable) feels a little clumsy. Properties are designed to alleviate this awkwardness.

Implementing encapsulation by using methods

First, let's recap the original motivation for using methods to hide fields. Consider the following structure, which represents a position on a computer screen as a pair of coordinates: *x* and *y*. Assume that the range of valid values for the *x*-coordinate is between 0 and 1279, and the range of valid values for the *y*-coordinate is between 0 and 1023.

```
struct ScreenPosition
{
    public int X;
    public int Y;

    public ScreenPosition(int x, int y)
    {
        this.X = rangeCheckedX(x);
        this.Y = rangeCheckedY(y);
    }

    private static int rangeCheckedX(int x)
    {
        if (x < 0 || x > 1279)
        {
            throw new ArgumentOutOfRangeException("X");
        }
        return x;
    }

    private static int rangeCheckedY(int y)
    {
        if (y < 0 || y > 1023)
        {
            throw new ArgumentOutOfRangeException("Y");
        }
        return y;
    }
}
```

One problem with this structure is that it doesn't follow the golden rule of encapsulation—it doesn't keep its data private. Public data is often a bad idea because the class cannot control the values that an application specifies. For example, the `ScreenPosition` constructor checks its parameters to ensure that they're in a specified range, but no such check can be done on the "raw" access to the public fields. Sooner or later (probably sooner), an error or misunderstanding on the part of a developer using this class in an application can cause either *X* or *Y* to stray out of this range:

```
ScreenPosition origin = new ScreenPosition(0, 0);
...
int xpos = origin.X;
origin.Y = -100; // oops
```

The common way to solve this problem is to make the fields private and add an accessor method and a modifier method to read and write the value of each private field, respectively. The modifier methods can then check the range for new field values. For example, the code that follows contains an accessor (GetX) and a modifier (SetX) for the X field. Notice that SetX checks the parameter passed in.

```
struct ScreenPosition
{
    private int X;
    private int Y;
    ...
    public int GetX()
    {
        return this.x;
    }

    public void SetX(int newX)
    {
        this.x = rangeCheckedX(newX);
    }
    ...
    private static int rangeCheckedX(int x) { ... }
    private static int rangeCheckedY(int y) { ... }
    private int x, y;
}
```

The code now successfully enforces the range constraints, which is good. However, there's a price to pay for this valuable guarantee: `ScreenPosition` no longer has a natural field-like syntax. It uses awkward method-based syntax instead.

The example that follows increases the value of X by 10. To do so, it must read the value of X by using the GetX accessor method and then write the value of X by using the SetX modifier method.

```
int xpos = origin.GetX();
origin.SetX(xpos + 10);
```

Compare this with the equivalent code if the X field were public:

```
origin.X += 10;
```

There's no doubt that in this case, using public fields is syntactically cleaner, shorter, and easier. Unfortunately, using public fields breaks encapsulation. By using properties, you can combine the best of both worlds—fields and methods—to retain encapsulation while providing a field-like syntax.

What are properties?

A *property* is a cross between a field and a method—it looks like a field but acts as a method. You access a property by using the same syntax that you use to access a field. However, the compiler automatically translates this field-like syntax into calls to accessor methods (sometimes referred to as *property getters* and *property setters*).

The syntax for a property declaration looks like this:

```
AccessModifier Type PropertyName
{
    get
    {
        // read accessor code
    }

    set
    {
        // write accessor code
    }
}
```

A property can contain two blocks of code, starting with the `get` and `set` keywords. The `get` block contains statements that execute when the property is read, and the `set` block contains statements that run when the property is written to. The type of the property specifies the type of data read and written by the `get` and `set` accessors.

The next code example shows the `ScreenPosition` structure rewritten by using properties. When looking at this code, notice the following:

- Lowercase `_x` and `_y` are private fields.
- Uppercase `X` and `Y` are public properties.
- All `set` accessors are passed the data to be written by using a hidden, built-in parameter named `value`.

```
struct ScreenPosition
{
    private int _x, _y;

    public ScreenPosition(int X, int Y)
    {
        this._x = rangeCheckedX(X);
        this._y = rangeCheckedY(Y);
    }

    public int X
    {
        get { return this._x; }
        set { this._x = rangeCheckedX(value); }
    }

    public int Y
    {
        get { return this._y; }
        set { this._y = rangeCheckedY(value); }
    }

    private static int rangeCheckedX(int x) { ... }
    private static int rangeCheckedY(int y) { ... }
}
```

In this example, a private field directly implements each property, but this is only one way to implement a property. All that's required is for a get accessor to return a value of the specified type. Such a value can easily be calculated dynamically rather than being simply retrieved from stored data, in which case there might be no need for a physical field.



Note Although the examples in this chapter show how to define properties for a structure, properties are equally applicable to classes; the syntax is the same.

For simple properties, you can use expression-bodied members rather than full-blown method syntax for get and set accessors. For example, you can simplify the X and Y properties shown in the previous example like this:

```
public int X
{
    get => this._x;
    set => this._x = rangeCheckedX(value);
}

public int Y
{
    get => this._y;
    set => this._y = rangeCheckedY(value);
}
```

Notice that you don't need to specify the return keyword for the get accessor; you simply provide an expression that is evaluated every time the property is read. This syntax is less verbose and arguably more natural, although functionally the properties perform the same task. It's a matter of personal preference as to which you should use, but for simple properties, I recommend adopting the expression-bodied syntax. Of course, you can mix and match; you could implement a simple get accessor as an expression-bodied member, but a more complex set accessor could still utilize the method syntax.

Properties and field names: a warning

The section "Naming variables" in Chapter 2, "Working with variables, operators, and expressions," offers recommendations for naming variables. In particular, it states that you should avoid starting an identifier with an underscore. However, you can see that the `ScreenPosition` structure doesn't completely follow this guidance; it contains fields named `_x` and `_y`.

There's a good reason for this anomaly. The sidebar "Naming and accessibility" in Chapter 7, "Creating and managing classes and objects," describes how it is common to use identifiers that start with an uppercase letter for publicly accessible methods and fields and to use identifiers that start with a lowercase letter for private methods and fields. Taken together, these two practices can result in you giving properties and private fields a name that differs only in the case of the initial letter, and many organizations do precisely this.

If your organization follows this approach, you should be aware of one important drawback. Examine the following code, which implements a class named `Employee`. The `employeeID` field is private, but the `EmployeeID` property provides public access to this field.

```
class Employee
{
    private int employeeID;

    public int EmployeeID
    {
        get => this.EmployeeID;
        set => this.EmployeeID = value;
    }
}
```

This code will compile perfectly well but results in a program raising a `StackOverflowException` exception whenever the `EmployeeID` property is accessed. The exception occurs because the `get` and `set` accessors reference the property (uppercase E) rather than the private field (lowercase e), which triggers an endless recursive loop that eventually causes the process to exhaust the available memory. This type of bug is very difficult to spot! For this reason, the examples in this book name the private fields used to provide the data for properties with a leading underscore; it makes them much easier to distinguish from the names of properties. All other private fields will continue to use camelCase identifiers without a leading underscore.

Using properties

When you use a property in an expression, you can use it in a read context (when you are retrieving its value) and in a write context (when you are modifying its value). The following example shows how to read values from the `X` and `Y` properties of the `ScreenPosition` structure:

```
ScreenPosition origin = new ScreenPosition(0, 0);
int xpos = origin.X;    // calls origin.X.get
int ypos = origin.Y;    // calls origin.Y.get
```

Notice that you access properties and fields by using identical syntax. When you use a property in a read context, the compiler automatically translates your field-like code into a call to the `get` accessor of that property. Similarly, if you use a property in a write context, the compiler automatically translates your field-like code into a call to the `set` accessor of that property.

```
origin.X = 40;          // calls origin.X.set, with value set to 40
origin.Y = 100;         // calls origin.Y.set, with value set to 100
```

The values being assigned are passed into the `set` accessors by using the `value` variable, as described in the preceding section. The runtime does this automatically.

It's also possible to use a property in a read/write context. In this case, both the `get` accessor and the `set` accessor are used. For example, the compiler automatically translates statements such as the following into calls to the `get` and `set` accessors:

```
origin.X += 10;
```



Tip You can declare static properties in the same way you do static fields and methods. You can access static properties by using the name of the class or structure rather than an instance of the class or structure.

Read-only properties

You can declare a property that contains only a get accessor. In this case, you can use the property only in a read context. For example, here's the `X` property of the `ScreenPosition` structure declared as a read-only property:

```
struct ScreenPosition
{
    private int _x;
    ...
    public int X
    {
        get => this._x;
    }
}
```

The `X` property does not contain a set accessor; therefore, any attempt to use `X` in a write context will fail, as demonstrated in the following example:

```
origin.X = 140; // compile-time error
```

Write-only properties

Similarly, you can declare a property that contains only a set accessor. In this case, you can use the property only in a write context. For example, here's the `X` property of the `ScreenPosition` structure declared as a write-only property:

```
struct ScreenPosition
{
    private int _x;
    ...
    public int X
    {
        set => this._x = rangeCheckedX(value);
    }
}
```

The `X` property doesn't contain a get accessor; therefore, any attempt to use `X` in a read context will fail, as illustrated here:

```
Console.WriteLine(origin.X); // compile-time error
origin.X = 200;              // compiles OK
origin.X += 10;              // compile-time error
```




Note Write-only properties are useful for secure data such as passwords. Ideally, an application that implements security should allow you to set your password but never allow you to read it back. When a user attempts to log on, the user can provide the password. The logon method can compare this password with the stored password and return only an indication of whether they match.

Property accessibility

You can specify the accessibility of a property (using the keywords `public`, `private`, or `protected`) when you declare it. However, it's possible within the property declaration to override the property accessibility for the `get` and `set` accessors. For example, the version of the `ScreenPosition` structure shown in the code that follows defines the `set` accessors of the `X` and `Y` properties as `private`. (The `get` accessors are `public` because the properties are `public`.)

```
struct ScreenPosition
{
    private int _x, _y;
    ...
    public int X
    {
        get => this._x;
        private set => this._x = rangeCheckedX(value);
    }

    public int Y
    {
        get => this._y;
        private set => this._y = rangeCheckedY(value);
    }
    ...
}
```

You must observe some rules when defining accessors that have different accessibility from one another:

- You can change the accessibility of only one of the accessors when you define it. It wouldn't make much sense to define a property as `public` only to change the accessibility of both accessors to `private` anyway.
- The modifier must not specify an accessibility that is less restrictive than that of the property. For example, if the property is declared to be `private`, you can't specify the read accessor as `public`. (Instead, you would make the property `public` and make the write accessor `private`.)

Understanding property restrictions

Properties look, act, and feel like fields when you read or write data by using them. However, they are not true fields, and certain restrictions apply to them:

- You can assign a value through a property of a structure or class only after the structure or class has been initialized. The following code example is illegal because the `location` variable has not been initialized (by using `new`):

```
ScreenPosition location;  
location.X = 40; // compile-time error; location variable not initialized
```



Note This might seem trivial, but if `X` were a field rather than a property, the code would be legal. For this reason, you should define structures and classes from the beginning by using properties rather than fields that you later migrate to properties. Code that uses your classes and structures might no longer work after you change fields into properties. You'll return to this topic in the section "Generating automatic properties" later in this chapter.

- You can't use a property as a `ref` or an `out` argument to a method (although you can use a writable field as a `ref` or an `out` argument). This makes sense because the property doesn't really point to a memory location; rather, it points to an accessor method, such as in the following example:

```
MyMethod(ref location.X); // compile-time error
```
- A property can contain at most one get accessor and one set accessor. A property cannot contain other methods, fields, or properties.
- The get and set accessors cannot take any parameters. The data being assigned is passed to the set accessor automatically by using the `value` variable.
- You can't declare properties by using `const`, as demonstrated here:

```
const int X  
{  
    get => ...  
    set => ...  
} // compile-time error
```

Using properties appropriately

Properties are a powerful feature. Used correctly, they can help to make code easier to understand and maintain. However, they're no substitute for careful object-oriented design that focuses on the behavior of objects rather than on the properties of objects.

Accessing private fields through regular methods or through properties does not, by itself, make your code well designed. For example, a bank account holds a balance indicating the funds available in the account. You might therefore be tempted to create a `Balance` property on a `BankAccount` class, like this:

```
class BankAccount
{
    private decimal _balance;
    ...
    public decimal Balance
    {
        get => this._balance;
        set => this._balance = value;
    }
}
```

This is a poor design because it fails to represent the functionality required when someone withdraws money from or deposits money into an account. (If you know of a bank that allows you to increase the balance of your account directly without physically putting money into the account, please let me know!)

When you're programming, try to express the problem you're solving in the solution and don't get lost in a mass of low-level syntax. As the following example illustrates, provide `Deposit` and `Withdraw` methods for the `BankAccount` class rather than a property setter:

```
class BankAccount
{
    private decimal _balance;
    ...
    public decimal Balance { get => this._balance; }
    public void Deposit(decimal amount) { ... }
    public bool Withdraw(decimal amount) { ... }
}
```

Declaring interface properties

You encountered interfaces in Chapter 13, “Creating interfaces and defining abstract classes.” You can use interfaces to define properties as well as methods. To do this, you specify the `get` or `set` keyword or both, but you replace the body of the `get` or `set` accessor with a semicolon, as shown here:

```
interface IScreenPosition
{
    int X { get; set; }
    int Y { get; set; }
}
```

Any class or structure that implements this interface must implement the `X` and `Y` properties with `get` and `set` accessor methods (or expression-bodied members).

```
struct ScreenPosition : IScreenPosition
{
    ...
    public int X
    {
        get { ... } // or get => ...
        set { ... } // or set => ...
    }

    public int Y
    {
        get { ... }
        set { ... }
    }
    ...
}
```

If you implement the interface properties in a class, you can declare the property implementations as `virtual`, which enables derived classes to override the implementations.

```
class ScreenPosition : IScreenPosition
{
    ...
    public virtual int X
    {
        get { ... }
        set { ... }
    }
    public virtual int Y
    {
        get { ... }
        set { ... }
    }
    ...
}
```



Note This example shows a class. Remember that the `virtual` keyword is not valid when creating a structure because structures do not support inheritance.

You can also choose to implement a property by using the explicit interface implementation syntax covered in Chapter 13. An explicit implementation of a property is nonpublic and nonvirtual (and cannot be overridden).

```
struct ScreenPosition : IScreenPosition
{
    ...
    int IScreenPosition.X
    {
        get { ... }
        set { ... }
    }

    int IScreenPosition.Y
    {
        get { ... }
        set { ... }
    }
    ...
}
```

Replacing methods with properties

Chapter 13 taught you how to create a drawing application with which a user can place circles and squares on a canvas in a window. In the exercises in that chapter, you factored the common functionality for the `Circle` and `Square` classes into an abstract class called `DrawingShape`. The `DrawingShape` class provides the `SetLocation` and `SetColor` methods, which the application uses to specify the position and color of a shape on the screen. In the following exercise, you'll modify the `DrawingShape` class to expose the location and color of a shape as properties.

To use properties

1. Start Visual Studio 2022, if it is not already running.
2. Open the **Drawing** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 15\Drawing Using Properties** folder in your **Documents** folder.
3. Display the `DrawingShape.cs` file in the Code and Text Editor window.

This file contains the `DrawingShape` class that is in Chapter 13 except that, following the recommendations described earlier in this chapter, the `size` field has been renamed `_size`, and the `locX` and `locY` fields have been renamed `_x` and `_y`.

```
abstract class DrawingShape
{
    protected int _size;
    protected int _x = 0, _y = 0;
    ...
}
```

4. Open the `IDraw.cs` file for the Drawing project in the Code and Text Editor window.

This interface specifies the `SetLocation` method, like this:

```
interface IDraw
{
    void SetLocation(int xCoord, int yCoord);
    ...
}
```

The purpose of this method is to set the `_x` and `_y` fields of the `DrawingShape` object to the values passed in. This method can be replaced with a pair of properties.

5. Delete this method and replace it with the definition of a pair of properties named `X` and `Y`, as shown here in bold:

```
interface IDraw
{
    int X { get; set; }
    int Y { get; set; }
    ...
}
```

6. In the `DrawingShape` class, delete the `SetLocation` method and replace it with the following implementations of the `X` and `Y` properties:

```
public int X
{
    get => this._x;
    set => this._x = value;
}

public int Y
{
    get => this._y;
    set => this._y = value;
}
```

7. Display the `DrawingPad.xaml.cs` file in the Code and Text Editor window and locate the `drawingCanvas_Tapped` method.

This method runs when a user taps the screen or clicks the left mouse button. It draws a square on the screen at the point where the user taps or clicks.

8. Locate the statement that calls the `SetLocation` method to set the position of the square on the screen. It's located in the `if` statement block, as shown in bold in the following:

```
if (mySquare is IDraw)
{
    IDraw drawSquare = mySquare;
    drawSquare.SetLocation((int)mouseLocation.X, (int)mouseLocation.Y);
    drawSquare.Draw(drawingCanvas);
}
```

9. Replace this statement with code that sets the X and Y properties of the Square object, as shown in bold in the following code:

```
if (mySquare is IDraw)
{
    IDraw drawSquare = mySquare;
    drawSquare.X = (int)mouseLocation.X;
    drawSquare.Y = (int)mouseLocation.Y;
    drawSquare.Draw(drawingCanvas);
}
```

10. Locate the `drawingCanvas_RightTapped` method.

This method runs when the user taps and holds a finger on the screen or clicks the right mouse button. It draws a circle at the location where the user taps and holds or right-clicks.

11. In this method, replace the statement that calls the `SetLocation` method of the `Circle` object and set the X and Y properties instead, as shown in bold in the following example:

```
if (myCircle is IDraw)
{
    IDraw drawCircle = myCircle;
    drawCircle.X = (int)mouseLocation.X;
    drawCircle.Y = (int)mouseLocation.Y;
    drawCircle.Draw(drawingCanvas);
}
```

12. Open the `IColor.cs` file for the Drawing project in the Code and Text Editor window.

This interface specifies the `SetColor` method, like this:

```
interface IColor
{
    void SetColor(Color color);
}
```

13. Delete this method and replace it with the definition of a property named `Color`, as presented here:

```
interface IColor
{
    Color Color { set; }
}
```

This is a write-only property, providing a set accessor but no get accessor. You define the property this way because the color isn't stored in the `DrawingShape` class and is specified only as each shape is drawn. You can't actually query a shape to find out which color it is.



Note It's common practice for a property to share the same name as a type (`Color` in this example).

14. Return to the `DrawingShape` class in the Code and Text Editor window. Replace the `SetColor` method in this class with the `Color` property shown here:

```
public Color Color
{
    set
    {
        if (this.shape is not null)
        {
            SolidColorBrush brush = new SolidColorBrush(value);
            this.shape.Fill = brush;
        }
    }
}
```



Note The code for the set accessor is almost the same as the original `SetColor` method except that the statement that creates the `SolidColorBrush` object is passed the `value` parameter. Additionally, this is an example where the method syntax is more appropriate than using an expression-bodied member.

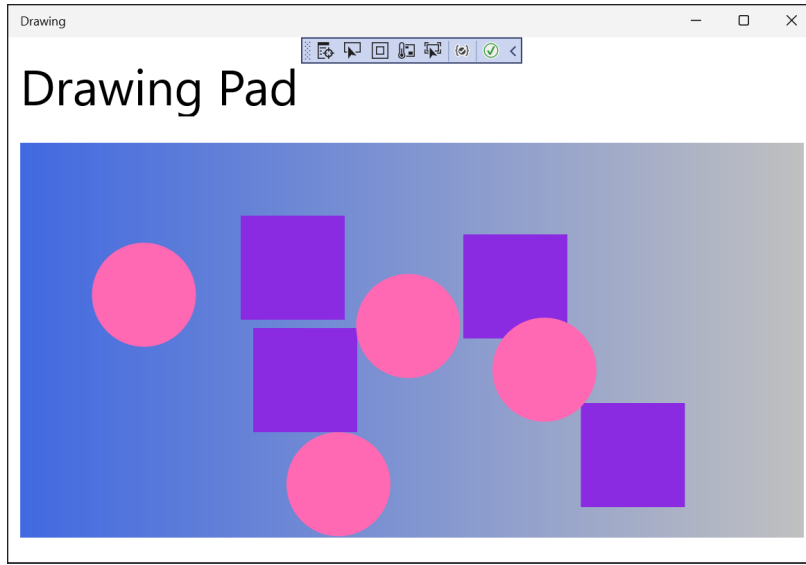
15. Return to the `DrawingPad.xaml.cs` file in the Code and Text Editor window. In the `drawingCanvas_Tapped` method, modify the statement that sets the color of the `Square` object to match the following code in bold:

```
if (mySquare is IColor)
{
    IColor colorSquare = mySquare;
    colorSquare.Color = Colors.BlueViolet;
}
```

16. Similarly, in the `drawingCanvas_RightTapped` method, modify the statement that sets the color of the `Circle` object as shown in bold.

```
if (myCircle is IColor)
{
    IColor colorCircle = myCircle;
    colorCircle.Color = Colors.HotPink;
}
```

17. On the **Debug** menu, select **Start Debugging** to build and run the project.
18. Verify that the application operates in the same manner as before. If you tap the screen or click the left mouse button on the canvas, the application should draw a square, and if you tap and hold or click the right mouse button, the application should draw a circle:



19. Return to the Visual Studio 2022 programming environment and stop debugging.

Pattern-matching with properties

C# includes pattern-matching capabilities that can take advantage of property values in a type. For example, suppose you're modeling bank accounts in an application. You define the following interface:

```
interface IBankAccount
{
    decimal Balance { get; }
    decimal OverdraftLimit { get; }
    void Deposit(decimal amount);
    decimal Withdraw(decimal amount);
}
```

You implement several types of bank accounts, each with varying limitations and interest rates:

```
class CurrentAccount : IBankAccount
{
    // Implementation details not shown
    ...
}

class DepositAccount : IBankAccount
{
    // Implementation details not shown
    ...
}

class HighInterestSavingsAccount : IBankAccount
{
    // Implementation details not shown
    ...
}
```

You want to write a method that calculates the interest receivable for any of these types of account. The amount of interest paid depends on the balance and the type of account. The `switch` statement enables you to perform a complex set of rules quickly and easily, based on pattern-matching capabilities that enable you to determine the type of an object and the values of its properties. The following code shows the `CalculateInterest` method:

```
public static decimal CalculateInterest(IBankAccount acc) =>
{
    acc switch
    {
        CurrentAccount ca when ca.Balance < 500 => 0.05m,
        CurrentAccount ca when ca.Balance >= 500 && ca.Balance < 5000 => 0.5m,
        CurrentAccount ca when ca.Balance >= 5000 => 0.75m,
        DepositAccount da when da.Balance >= 2000 => 0.75m,
        DepositAccount => 0.5m,
        HighInterestSavingsAccount hisa when hisa.Balance >= 5000 => 2.0m,
        HighInterestSavingsAccount => 1.2m,
        _ => throw new ArgumentException($"Unknown account type")
    };
}
```

The `when` clause for each `switch` expression examines the `Balance` property. This approach is clear and concise; it reads like a user specification of how the interest should be calculated rather than a block of C# code.

Generating automatic properties

As mentioned earlier in this chapter, the principal purpose of properties is to hide the implementation of fields from the outside world. This is fine if your properties actually perform some useful work, but if the `get` and `set` accessors simply wrap operations that just read or assign a value to a field, you might be questioning the value of this approach.

There are at least two good reasons why you should define properties rather than expose data as public fields even in these situations:

- **Compatibility with applications** Fields and properties expose themselves by using different metadata in assemblies. If you develop a class and decide to use public fields, any applications that use the class will reference these items as fields. Although you use the same C# syntax for reading and writing a field that you use when reading and writing a property, the compiled code is actually quite different—the C# compiler just hides the differences from you. If you later decide that you really do need to change these fields to properties (maybe business requirements have changed, and you need to perform additional logic when assigning values), existing applications will not be able to use the updated version of the class without being recompiled. This is awkward if you've deployed the application on a large number of devices throughout an organization. There are ways around this, but it's generally better to avoid getting into this situation in the first place.
- **Compatibility with interfaces** If you're implementing an interface and the interface defines an item as a property, you must write a property that matches the specification in the interface, even if the property just reads and writes data in a private field. You can't implement a property simply by exposing a public field with the same name.

The designers of the C# language recognized that programmers are busy people who should not have to waste their time writing more code than they need to. To this end, the C# compiler can generate the code for properties automatically, like this:

```
class Circle
{
    public int Radius{ get; set; }
    ...
}
```

In this example, the `Circle` class contains a property named `Radius`. Apart from the type of this property, you have not specified how this property works—the `get` and `set` accessors are empty. The C# compiler converts this definition to a private field and a default implementation that looks similar to this:

```
class Circle
{
    private int _radius;
    public int Radius{
        get
        {
            return this._radius;
        }

        set
        {
            this._radius = value;
        }
    }
    ...
}
```

So, with very little effort, you can implement a simple property by using automatically generated code. If you need to include additional logic later, you can do so without breaking any existing applications.



Note The syntax for defining an automatic property is almost identical to the syntax for defining a property in an interface. The exception is that an automatic property can specify an access modifier such as `private`, `public`, or `protected`.

You can create a read-only automatic property by omitting the empty `set` accessor from your property declaration, like this:

```
class Circle
{
    public DateTime CircleCreatedDate { get; }
    ...
}
```

This is useful in scenarios where you want to create an immutable property—that is, a property that is set when the object is constructed and cannot subsequently be changed. For example, you might want to set the date on which an object was created or the name of the user who created it, or you might want to generate a unique identifier value for the object. These are values that you typically want to set once and then prevent them from being modified. With this in mind, C# allows you to initialize

read-only automatic properties in one of two ways. You can initialize the property from a constructor, like this:

```
class Circle
{
    public Circle()
    {
        CircleCreatedDate = DateTime.Now;
    }

    public DateTime CircleCreatedDate { get; }
    ...
}
```

Alternatively, you can initialize the property as part of the declaration, like this:

```
class Circle
{
    public DateTime CircleCreatedDate { get; } = DateTime.Now;
    ...
}
```

Be aware that if you initialize a property in this way and also set its value in a constructor, the value provided by the constructor will overwrite the value specified by the property initializer. Use one approach or the other, but not both!



Note You cannot create write-only automatic properties. If you attempt to create an automatic property without a get accessor, you'll see a compile-time error.

Initializing objects by using properties

In Chapter 7, you learned how to define constructors to initialize an object. An object can have multiple constructors, and you can define constructors with varying parameters to initialize different elements in an object. For example, you could define a class that models a triangle, like this:

```
public class Triangle
{
    private int side1Length;
    private int side2Length;
    private int side3Length;

    // default constructor - default values for all sides
    public Triangle()
    {
        this.side1Length = this.side2Length = this.side3Length = 10;
    }

    // specify length for side1Length, default values for the others
    public Triangle(int length1)
    {
        this.side1Length = length1;
    }
}
```

```

        this.side2Length = this.side3Length = 10;
    }

    // specify length for side1Length and side2Length,
    // default value for side3Length
    public Triangle(int length1, int length2)
    {
        this.side1Length = length1;
        this.side2Length = length2;
        this.side3Length = 10;
    }

    // specify length for all sides
    public Triangle(int length1, int length2, int length3)
    {
        this.side1Length = length1;
        this.side2Length = length2;
        this.side3Length = length3;
    }
}

```

Depending on how many fields a class contains and the various combinations you want to enable for initializing the fields, you could end up writing a lot of constructors. There are also potential problems if many of the fields have the same type: You might not be able to write a unique constructor for all combinations of fields. For example, in the preceding `Triangle` class, you could not easily add a constructor that initializes only the `side1Length` and `side3Length` fields because it wouldn't have a unique signature; it would take two `int` parameters, and the constructor that initializes `side1Length` and `side2Length` already has this signature.

One possible solution is to define a constructor that takes optional parameters and specify values for the parameters as named arguments when you create a `Triangle` object. However, a better and more transparent solution is to initialize the private fields to a set of default values and expose them as properties, like this:

```

public class Triangle
{
    private int side1Length = 10;
    private int side2Length = 10;
    private int side3Length = 10;

    public int Side1Length
    {
        set => this.side1Length = value;
    }

    public int Side2Length
    {
        set => this.side2Length = value;
    }

    public int Side3Length
    {
        set => this.side3Length = value;
    }
}

```

When you create an instance of a class, you can initialize it by specifying the names and values for any public properties that have set accessors. For example, you can create `Triangle` objects and initialize any combination of the three sides, like this:

```
Triangle tri1 = new Triangle { Side3Length = 15 };
Triangle tri2 = new Triangle { Side1Length = 15, Side3Length = 20 };
Triangle tri3 = new Triangle { Side2Length = 12, Side3Length = 17 };
Triangle tri4 = new Triangle { Side1Length = 9, Side2Length = 12, Side3Length = 15 };
```

This syntax is known as an *object initializer*. When you invoke an object initializer in this way, the C# compiler generates code that calls the default constructor and then calls the set accessor of each named property to initialize it with the value specified. You can specify object initializers in combination with non-default constructors as well. For example, if the `Triangle` class also provided a constructor that took a single string parameter describing the type of triangle, you could invoke this constructor and initialize the other properties, like this:

```
Triangle tri5 = new Triangle("Equilateral triangle")
{
    Side1Length = 3,
    Side2Length = 3,
    Side3Length = 3
};
```

The important point to remember is that the constructor runs first, and the properties are set afterward. Understanding this sequencing is important if the constructor sets fields in an object to specific values, and the properties that you specify change these values.

Automatic properties and immutability

One drawback with automatic properties is that they must be mutable to allow initializers to function correctly. Recall that the object initialization process works by first instantiating the object by using a constructor and then writing values to the specified properties once the object has been created. If a property is read-only (it has a get accessor but no set accessor), then any attempt to change property values once the object has been instantiated will fail, even using object initialization syntax.

Fortunately, C# 9.0 and later provides automatic init-only properties to handle this situation. You create an init-only property by using an `init` accessor. The following example shows a class that captures student grade information for a school:

```
class Grade
{
    public int StudentID { get; init; }
    public string Subject { get; init; }
    public char SubjectGrade { get; init; }
}
```

You can create a new `Grade` object and initialize its values as before:

```
var grade1 = new Grade() { StudentID = 1, Subject = "Math", SubjectGrade = 'A' };
```

However, any attempt to modify any fields in the object will result in a compiler error:

```
grade1.SubjectGrade = 'B'; // Causes the compiler error:  
// Init-only property or indexer 'Grade.SubjectGrade'  
// can only be assigned in an object initializer
```

In the next exercise, you'll define a class for modeling regular polygons that contains automatic properties for providing access to information about the number of sides the polygon contains and the length of these sides. In this exercise, you'll implement `get` and `set` accessors. You'll see `init` accessors in action in the final part of this chapter.

To define automatic properties and use object initializers

1. In Visual Studio 2022, open the **AutomaticProperties** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 15\AutomaticProperties** folder in your **Documents** folder.

The **AutomaticProperties** project contains the **Program.cs** file, defining the **Program** class with the **Main** and **doWork** methods that you've seen in previous exercises.

2. In Solution Explorer, right-click the **AutomaticProperties** project, point to **Add**, and then select **Class** to open the Add New Item – AutomaticProperties dialog.

3. In the **Name** box, type **Polygon.cs**, and then select **Add**.

The **Polygon.cs** file, holding the **Polygon** class, is created and added to the project and appears in the Code and Text Editor window.

4. Add the automatic properties **NumSides** and **SideLength** to the **Polygon** class, as shown here in bold:

```
class Polygon  
{  
    public int NumSides { get; set; }  
    public double SideLength { get; set; }  
}
```

5. Add the following default constructor shown in bold to the **Polygon** class:

```
class Polygon  
{  
    ...  
    public Polygon()  
    {  
        this.NumSides = 4;  
        this.SideLength = 10.0;  
    }  
}
```

This constructor initializes the **NumSides** and **SideLength** fields with default values. In this exercise, the default polygon is a square with sides 10 units long.

6. Display the Program.cs file in the Code and Text Editor window.
7. Add the statements shown here in bold to the doWork method, replacing the `// TODO: comment:`

```
static void doWork()
{
    Polygon square = new Polygon();
    Polygon triangle = new Polygon { NumSides = 3 };
    Polygon pentagon = new Polygon { SideLength = 15.5, NumSides = 5 };
}
```

These statements create `Polygon` objects. The `square` variable is initialized by using the default constructor. The `triangle` and `pentagon` variables are also initialized by using the default constructor, and then this code changes the value of the properties exposed by the `Polygon` class. In the case of the `triangle` variable, the `NumSides` property is set to 3, but the `SideLength` property is left at its default value of 10.0. For the `pentagon` variable, the code changes the values of the `SideLength` and `NumSides` properties.

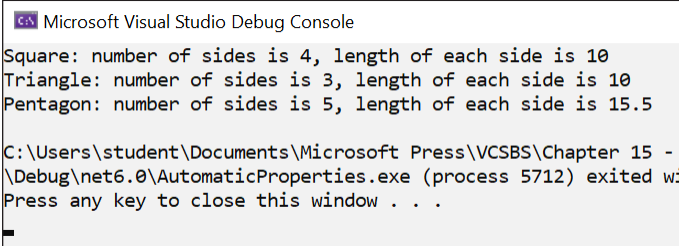
8. Add the following code shown in bold to the end of the `doWork` method:

```
static void doWork()
{
    ...
    Console.WriteLine($"Square: number of sides is {square.NumSides}, length of each side
is {square.SideLength}");
    Console.WriteLine($"Triangle: number of sides is {triangle.NumSides}, length of each
side is {triangle.SideLength}");
    Console.WriteLine($"Pentagon: number of sides is {pentagon.NumSides}, length of each
side is {pentagon.SideLength}");
}
```

These statements display the values of the `NumSides` and `SideLength` properties for each `Polygon` object.

9. On the **Debug** menu, select **Start Without Debugging**.

Verify that the program builds and runs, writing the messages shown here to the console window.



```
Microsoft Visual Studio Debug Console
Square: number of sides is 4, length of each side is 10
Triangle: number of sides is 3, length of each side is 10
Pentagon: number of sides is 5, length of each side is 15.5

C:\Users\student\Documents\Microsoft Press\VCSBS\Chapter 15 - 
\Debug\net6.0\AutomaticProperties.exe (process 5712) exited wi
Press any key to close this window . . .
```

10. Press **Enter** to close the application and return to Visual Studio 2022.

Using records with properties to implement lightweight structures

You create structures and classes to model entities. The fields in an entity represent the state of the entity. In many cases, this state may change over time. However, there will likely be situations where you want an entity to be immutable.

You've seen how you can create structures and classes with read-only and init-only properties to provide this immutability. However, the syntax for creating such a type can be a little long-winded. When you define an immutable object, you really just want to specify the names and types of its fields and get the compiler to generate the code that provides the implementation for you. Records, which were introduced in C# 9.0, provide this capability.

Think of a record as a lightweight structure. A record is a value type, with built-in capabilities to support immutability. Another feature of records is their notion of equality. When you compare two objects for equality, what are you actually doing? For classes, equality verifies that objects are actually references to the same item on the heap. For structures, equality is value-based; two structures are equal if they have fields that contain the same data. However, the default implementation of equality for a structure is through the `Equals` method inherited from `ValueType`. This implementation uses reflection to query the fields in the structures being compared and then performs a field-by-field check across the two structures.

This approach is very generalized, and although it works, it is not efficient. You can override the `Equals` method to provide your own version, but if you do so, you should also override the `GetHashCode` method. This method is called by various methods in the .NET libraries to ascertain the ID of an object, and no two instances of a structure should return the same value. The overall result is that an ostensibly simple structure can end up ballooning in size to provide the necessary coding infrastructure required to use the structure.



Note Structures created in Universal Windows Platform (UWP) applications and UWP class libraries aren't actually based on the `ValueType` base type. UWP apps use a different runtime that generates default implementations of these methods.

Returning to the student grades example from earlier, you could implement the `Grade` type as a structure, like this (some details have been omitted):

```
struct Grade
{
    public int StudentID { get; init; }
    public string Subject { get; init; }
    public char SubjectGrade { get; init; }
    public override bool Equals(object obj)
    {
        ...
    }
}
```

```

        public override int GetHashCode()
        {
            ...
        }
    }
}

```

You can instantiate new instances of the `Grade` structure using object initialization as before, and you can copy data between instances:

```

var grade1 = new Grade() { StudentID = 1, Subject = "Math", SubjectGrade = 'A' };
var grade2 = new Grade() { StudentID = 1, Subject = "French", SubjectGrade = 'C' };
var grade3 = grade1; // Copy the grade1 instance

```

You can attempt to display the contents of a `Grade` structure:

```

Console.WriteLine(grade1);
Console.WriteLine(grade2);
Console.WriteLine(grade3);

```

The problem is that unless you have overridden the `ToString` method of the structure as described earlier in this book, the output simply looks like this:

```

Grade
Grade
Grade

```

To test for equality, you use the `Equals` method provided by your structure.

```

Console.WriteLine($"{grade1.Equals(grade3)}");

```

The output of this statement is:

```

True

```

The student grade type is a natural candidate for a record. You can define a record using a very similar syntax to that of a structure:

```

record Grade
{
    public int StudentID { get; init; }
    public string Subject { get; init; }
    public char SubjectGrade { get; init; }
}

```

However, records provide a neat shortcut syntax for achieving the same thing. You can define the same record type simply as:

```

record Grade(int StudentID, string Subject, char SubjectGrade);

```

Each field is generated with a `get` accessor and an `init` accessor. Furthermore, a constructor is created that enables you to specify values for each field. You can create instances of the `Grade` record and copy `Grade` instances, like this:

```

var grade1 = new Grade(1, "Math", 'A');
var grade2 = new Grade(1, "French", 'C');
var grade3 = grade1; // Copy the grade1 instance

```

If you feel this syntax is a little too concise, you can name each parameter provided to the constructor:

```
var grade1 = new Grade(StudentID:1, Subject:"Math", SubjectGrade:'A');
```

The real joy of records comes when you discover that they have a built-in implementation of the `ToString` method that displays the contents of the record automatically. The following statements:

```
Console.WriteLine(grade1);  
Console.WriteLine(grade2);  
Console.WriteLine(grade3);
```

display the output shown here:

```
Grade { StudentID = 1, Subject = Math, SubjectGrade = A }  
Grade { StudentID = 1, Subject = French, SubjectGrade = C }  
Grade { StudentID = 1, Subject = Math, SubjectGrade = A }
```

Additionally, a record type overrides the equality operator, `==`, enabling you to write code such as the following statement, which displays the value `True`:

```
Console.WriteLine($"{grade1 == grade3}");
```



Note Chapter 22, “Operator overloading,” describes how to overload operators in your own types.

Another trick provided by records is the ability to partially copy data from one record to another. If you wanted to create another `Grade` record with an A for Math but for a different student, you could use the following code:

```
var grade4 = grade1 with { StudentID = 2 };
```

Record types support deconstruction to enable you to quickly retrieve the field values from a record. You create a tuple with variables to match each of the fields, and then assign the record to the tuple. The compiler can infer the type of each variable in the tuple automatically. In the `Grade` example, you can deconstruct a record and display the field values using a tuple like this:

```
var (studentID, subjectName, grade) = grade1;  
Console.WriteLine($"{studentID}, {subjectName}, {grade}");
```

In the final exercise in this chapter, you’ll create a record type that captures student enrollment information when they sign up for a class in school.

To define and use a record type

1. In Visual Studio 2022, open the **StudentEnrollment** solution, which is located in the **\Microsoft Press\VCBS\Chapter 15\StudentEnrollment** folder in your **Documents** folder.

This project contains the `Program.cs` file, with the `Main` and `doWork` test harness methods that you’ve seen before.

2. In Solution Explorer, right-click the **StudentEnrollment** project, point to **Add**, and then select **Class** to open the Add New Item – StudentEnrollment dialog.

3. In the **Name** box, type **Enrollment.cs**, and then select **Add**.



Note There is currently no Record item template provided with Visual Studio, so you'll use the Class template and change it.

4. In the Enrollment.cs file, replace the Enrollment class with the following record definition:

```
record Enrollment(int StudentID, string CourseName, DateOnly DateEnrolled);
```

This is all you need to do. The compiler will generate init-only properties for each field, a constructor that populates each field together with the equality operator, and an override of the ToString method that displays the fields in the record.

5. Open the Program.cs file in the Code and Text Editor window.
6. In the doWork method, replace the // TODO: comment with the following code, which creates an array of four Enrollment records. The third record is a copy of the first. The fourth record is a copy of the first but with a different student ID:

```
static void doWork()
{
    var Enrollments = new Enrollment[4];
    Enrollments[0] = new Enrollment(StudentID: 1, CourseName: "Physics",
        DateEnrolled: new DateOnly(2021, 07, 20));
    Enrollments[1] = new Enrollment(StudentID: 1, CourseName: "Chemistry",
        DateEnrolled: new DateOnly(2021, 07, 20));
    Enrollments[2] = Enrollments[0];
    Enrollments[3] = Enrollments[0] with { StudentID = 2};
}
```

7. Add the following statements in bold to the end of the doWork method. These statements display the contents of each Enrollment record:

```
static void doWork()
{
    ...
    foreach (var Enrollment in Enrollments)
    {
        Console.WriteLine($"{Enrollment}");
    }
}
```

8. Append the following code in bold to the end of the doWork method. These statements compare the first Enrollment record in the Enrollments array to the other three and display the result of the comparison:

```
static void doWork()
{
    ...
    var firstEnrollment = Enrollments[0];
    foreach (var Enrollment in Enrollments[1..4])
    {
        Console.WriteLine($"{firstEnrollment == Enrollment}");
    }
}
```

9. Start the solution without debugging. Verify that the results look like this:

```
Enrollment { StudentID = 1, CourseName = Physics, DateEnrolled = 20/07/2021 }
Enrollment { StudentID = 1, CourseName = Chemistry, DateEnrolled = 20/07/2021 }
Enrollment { StudentID = 1, CourseName = Physics, DateEnrolled = 20/07/2021 }
Enrollment { StudentID = 2, CourseName = Physics, DateEnrolled = 20/07/2021 }
False
True
False
```

The comparisons show that `Enrollments[0]` matches `Enrollments[2]`, but not `Enrollments[1]` or `Enrollments[3]`. Remember that records are value types, so the comparison is based on the contents of the records. They are not the same physical record.

10. Return to Visual Studio.

11. To test the immutability of these records, add the following statement to the `doWork` method:

```
static void doWork()
{
    ...
    Enrollments[0].DateEnrolled = new DateOnly(2021, 08, 15);
}
```

12. On the **Build** menu, select **Rebuild Solution**. The build should fail with the message:

Init-only property or indexer 'Enrollment.DateEnrolled' can only be assigned in an object initializer, or on 'this' or 'base' in an instance constructor or an 'init' accessor.

13. Delete the statement and rebuild the project. It should build successfully again.

Summary

In this chapter, you saw how to create and use properties to provide controlled access to data in an object. You also saw how to create automatic properties and how to use properties when initializing objects. Finally, you learned how to use init-only properties with records to create immutable value types.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 16, “Handling binary data and using indexers.”
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

| To | Do this |
|---|--|
| Declare a read/write property for a structure or class | Declare the type of the property, its name, a get accessor, and a set accessor. For example:
<pre>struct ScreenPosition { ... public int X { get { ... } // or get => ... set { ... } // or set => ... } ... }</pre> |
| Declare a read-only property for a structure or class | Declare a property with only a get accessor. For example:
<pre>struct ScreenPosition { ... public int X { get { ... } // or get => ... } ... }</pre> |
| Declare a write-only property for a structure or class | Declare a property with only a set accessor. For example:
<pre>struct ScreenPosition { ... public int X { set { ... } // or set => ... } ... }</pre> |
| Declare a property in an interface | Declare a property with the get or set keyword or both. For example:
<pre>interface IScreenPosition { int X { get; set; } // no body int Y { get; set; } // no body }</pre> |
| Implement an interface property in a structure or class | In the class or structure that implements the interface, declare the property and implement the accessors. For example:
<pre>struct ScreenPosition : IScreenPosition { public int X { get { ... } set { ... } } public int Y { get { ... } set { ... } } }</pre> |

| To | Do this |
|--|--|
| Create an automatic property | <p>In the class or structure that contains the property, define the property with empty get and set accessors. For example:</p> <pre>class Polygon { public int NumSides { get; set; } }</pre> <p>If the property is read-only, then initialize the property either in the object constructor or as the property is defined. For example:</p> <pre>class Circle { public DateTime CircleCreatedDate { get; } = DateTime.Now; ... }</pre> |
| Use properties to initialize an object | <p>Specify the properties and their values as a list enclosed in braces when constructing the object. For example:</p> <pre>Triangle tri3 = new Triangle { Side2Length = 12, Side3Length = 17 };</pre> |
| Create immutable types | <p>Define init-only properties. For example:</p> <pre>class Grade { public int StudentID { get; init; } public string Subject { get; init; } public char SubjectGrade { get; init; } }</pre> |
| Instantiate an immutable type | <p>Use object initialization syntax and specify a value for each init-only property. For example:</p> <pre>var grade1 = new Grade() { StudentID = 1, Subject = "Math", SubjectGrade = 'A' };</pre> |
| Define and use an immutable record type | <p>Define the record and specify its properties. By default, the properties will be immutable. For example:</p> <pre>record Enrollment(int StudentID, string CourseName, DateOnly DateEnrolled);</pre> <p>Create new records using the default constructor generated for the record. For example:</p> <pre>var Enrollments = new Enrollment(StudentID: 1, CourseName: "Physics", DateEnrolled: new DateOnly(2021, 07, 20));</pre> |
| Retrieve the field values from a record type | <p>Deconstruct the record using a tuple. For example:</p> <pre>var (studentID, subjectName, grade) = grade1;</pre> |

Handling binary data and using indexers

After completing this chapter, you will be able to:

- Explain the purpose of indexers and create and use indexers to encapsulate logical array-like access to an object.
- Control read and write access to indexers by declaring accessors.
- Create interfaces that declare indexers.
- Implement indexers in structures and classes that inherit from interfaces.

Chapter 15, “Implementing properties to access fields,” described how to implement and use properties as a means of providing controlled access to the fields in a class. Properties are useful for mirroring fields that contain a single value. However, indexers are invaluable if you want to provide access to items that contain multiple values, and to do so by using a natural and familiar syntax.

What is an indexer?

You can think of an *indexer* as a smart array, in much the same way that you can think of a property as a smart field. Whereas a property encapsulates a single value in a class, an indexer encapsulates a set of values. The syntax that you use for an indexer is the same as the syntax that you use for an array.

The best way to understand indexers is to work through an example. First, you’ll consider a problem and examine a solution that doesn’t use indexers. Then you’ll work through the same problem and look at a better solution that does use indexers. The problem concerns integers, or more precisely, the `int` type. The example uses C# integers to store and query data stored as binary data, so it helps to understand how you can use the integer types in C# to store and manipulate binary values. We’ll discuss this first.

Storing binary values

You normally use an `int` to hold an integer value. Internally, an `int` stores its value as a sequence of 32 bits, where each bit can be either 0 or 1. Most of the time, you don't care about this internal binary representation; you just use an `int` type as a container to hold an integer value. Sometimes, however, programmers use the `int` type for other purposes. For example, some programs use an `int` as a set of binary flags and manipulate the individual bits within an `int`. If you're an old C hack like I am, what follows should feel very familiar.



Note Some older programs used `int` types to save memory. Such programs typically date from when the size of computer memory was measured in kilobytes rather than the gigabytes available these days and memory cost was at an absolute premium. A single `int` holds 32 bits, each of which can be 1 or 0. In some cases, programmers assigned 1 to indicate the value `true` and 0 to indicate `false`, and then employed an `int` as a set of Boolean values.

To make life a little easier for handling data that you want to treat as a collection of binary values, C# enables you to specify integer constants using binary notation. You indicate that a constant should be treated as a binary representation by prefixing it with `0b0`. For example, the following code assigns the binary value `1111` (15 in decimal) to a variable:

```
uint binData = 0b01111;
```

Note that this is a 4-bit value, but an integer occupies 32 bits. Any bits not specified are initialized to 0. You should also observe that when you specify an integer as a binary value, it's good practice to store the result as an unsigned int (`uint`). In fact, if you provide a full 32-bit binary value, the C# compiler will insist that you use an `uint`.

To help cope with long strings of bits, you can also insert the underscore (`_`) character as a separator between blocks of digits, like this:

```
uint moreBinData = 0b0_11110000_01011010_11001100_00001111;
```

In this example, the `_` separator is used to mark the byte boundaries (32 bits is 4 bytes). You can use the `_` separator anywhere within a binary constant (not just on byte boundaries); it is ignored by the C# compiler and is provided simply to help improve the readability of your code.

If you find binary strings a little lengthy, you can opt to specify values using hexadecimal (base 16) notation by using the `0x0` prefix. The following two statements assign the same values shown in the previous example to another pair of variables. Again, you can use the `_` character as a separator to make the values easier to read:

```
uint hexData = 0x0_0F;  
uint moreHexData = 0x0_F0_5A_CC_0F;
```

Displaying binary values

If you need to display the binary representation of an integer, you can use the `Convert.ToString` method. `Convert.ToString` is a heavily overloaded method that can generate a string representation of a range of data values held in different types. If you're converting integer data, you can additionally specify a numeric base (2, 8, 10, or 16), and the method will convert the integer to that base using an algorithm a little like the exercises you've seen in some earlier chapters. The following example prints out the binary value of the `moreHexData` variable:

```
uint moreHexData = 0x0_F0_5A_CC_0F;
Console.WriteLine($"{Convert.ToString(moreHexData, 2)}");
// displays 11110000010110101100110000001111
```

Manipulating binary values

C# provides a set of operators that you can use to access and manipulate the individual bits in an `uint`. These operators are as follows:

- **The NOT (~) operator** This is a unary operator that performs a bitwise complement. For example, if you apply the ~ operator to the 8-bit value `0b0_11001100` (204 decimal), you obtain the result `0b0_00110011` (51 decimal). All the ones in the original value become zeros, and all the zeros become ones.



Note The examples shown here are purely illustrative and are accurate only to 8 bits. In C#, the `int` type is 32 bits, so if you try any of these examples in a C# application, you will get a 32-bit result that might be different from those shown in this list. For example, in 32 bits, 204 is `0b0_00000000_00000000_00000000_11001100`, so in C#, `~204` is `0b0_11111111_11111111_11111111_00110011` (which is actually the `int` representation of `-205` in C#).

- **The left-shift (<<) operator** This is a binary operator that performs a left shift. The 8-bit expression `204 << 2` returns the value 48. (In binary, 204 decimal is `0b0_11001100`, and shifting it left by two places yields `0b0_00110000`, or 48 decimal.) The far-left bits are discarded, and zeros are introduced from the right. There is a corresponding right-shift operator (`>>`).
- **The OR (|) operator** This is a binary operator that performs a bitwise OR operation, returning a value containing a one in each position in which either of the operands has a one. For example, the 8-bit expression `204 | 24` has the value 220 (204 is `0b0_11001100`, 24 is `0b0_00011000`, and 220 is `0b0_11011100`).
- **The AND (&) operator** This operator performs a bitwise AND operation. AND is similar to the bitwise OR operator, but it returns a value containing a one in each position where both of the operands have a one. So, the 8-bit expression `204 & 24` is 8 (204 is `0b0_11001100`, 24 is `0b0_00011000`, and 8 is `0b0_00001000`).

- **The XOR (^) operator** This operator performs a bitwise exclusive OR operation, returning a one in each bit where there is a one in one operand or the other but not both. (Two ones yield a zero; this is the “exclusive” part of the operator.) So, the 8-bit expression `204 ^ 24` is 212 (`0b0_11001100 ^ 0b0_00011000` is `0b0_11010100`).

You can use these operators together to determine the values of the individual bits in an `int`. As an example, the following expression uses the left-shift (`<<`) and bitwise AND (`&`) operators to determine whether the sixth bit from the right of the byte variable named `bits` is set to 0 or to 1:

```
(bits & (1 << 5)) != 0
```



Note The bitwise operators count the positions of bits from right to left, and the bits are numbered starting at 0. So, bit 0 is the rightmost bit, and the bit at position 5 is the bit six places from the right.

Suppose the `bits` variable contains the decimal value 42. In binary, this is `0b0_00101010`. The decimal value 1 is `0b0_00000001` in binary, and the expression `1 << 5` has the value `0b0_00100000`; the sixth bit is 1. In binary, the expression `bits & (1 << 5)` is `0b0_00101010 & 0b0_00100000`, and the value of this expression is `0b0_00100000`, which is nonzero. If the variable `bits` contains the value 65, or `0b0_01000001`, the value of the expression is `0b0_01000001 & 0b0_00100000`, which yields the result `0b0_00000000`, or 0.

This is a fairly complicated example, but it’s trivial in comparison to the following expression, which uses the compound assignment operator `&=` to set the bit at position 6 to 0:

```
bits &= ~(1 << 5)
```

Similarly, if you want to set the bit at position 6 to 1, you can use the bitwise OR (`|`) operator. The following complicated expression is based on the compound assignment operator `|=`:

```
bits |= (1 << 5)
```

The trouble with these examples is that although they work, they are fiendishly difficult to understand. They’re complicated, and the solution is a very low-level one: it fails to create an abstraction of the problem that it solves, and it’s consequently very difficult to maintain code that performs these kinds of operations.

Solving the same problems using indexers

Let’s pull back from the preceding low-level solution for a moment and remember what the problem is: you’d like to use an `int` not as an `int` but as an array of bits. Therefore, the best way to solve this problem is to use an `int` as if it were an array of bits. For instance, to gain access and write a binary value to any one of the six positions in the `bits` variable, from the right, use an expression such as the following (remember that arrays start with index 0):

```
bits[5];
```

And, to set the bit four places from the right to `true`, you'd like to be able to write this:

```
bits[3] = true; // currently invalid code
```



Note To seasoned C developers, the Boolean value `true` is synonymous with the binary value 1, and the Boolean value `false` is synonymous with the binary value 0. Consequently, in C, the expression `bits[3] = true` means “Set the bit four places from the right of the `bits` variable to 1.”

Unfortunately, you can't use the square bracket notation on an `int`; it works only on an array or on a type that behaves like an array. So, the solution to the problem is to create a new type that acts, feels, and is used like an array of `bool` variables but is implemented by using an `int`. You can achieve this feat by defining an indexer.

Let's call this new type `IntBits`. `IntBits` will contain an `int` value (initialized in its constructor), but the idea is that you'll use `IntBits` as an array of `bool` variables.



Tip The `IntBits` type is small and lightweight, so it makes sense to create it as a structure rather than as a class.

```
struct IntBits
{
    private int bits;

    // Simple constructor, implemented as an expression-bodied method
    public IntBits(int initialBitValue) => bits = initialBitValue;

    // indexer to be written here
}
```

To define the indexer, you use a notation that is a cross between a property and an array. You introduce the indexer with the `this` keyword, specify the type of the value returned by the indexer, and specify the type of the value to use as the index into the indexer between square brackets. The indexer for the `IntBits` struct uses an integer as its index type and returns a Boolean value. It looks like this:

```
struct IntBits
{
    ...
    public bool this [ int index ]
    {
        get => (bits & (1 << index)) != 0;

        set
        {
            if (value) // turn the bit on if value is true; otherwise, turn it off
                bits |= (1 << index);
            else
                bits &= ~(1 << index);
        }
    }
}
```

Notice the following points:

- An indexer is not a method. There are no parentheses containing a parameter, but there are square brackets that specify an index. This index is used to specify which element is being accessed.
- All indexers use the `this` keyword. A class or structure can define at most one indexer (although you can overload it and have several implementations), and it is always named `this`.
- Indexers contain `get` and `set` accessors just like properties. In this example, the `get` and `set` accessors contain the complicated bitwise expressions discussed previously.
- The index specified in the indexer declaration is populated with the index value specified when the indexer is called. The `get` and `set` accessor methods can read this argument to determine which element should be accessed.



Note You should perform a range check on the index value in the indexer to prevent any unexpected exceptions from occurring in your indexer code.

It's also good practice to provide a way to display the data in this structure. You can do this by overriding the `ToString` method and converting the value held in the structure to a string containing its binary representation, like this:

```
struct IntBits
{
    ...
    public override string ToString()
    {
        return (Convert.ToString(bits, 2);
    }
}
```

After you've created the indexer, you can use a variable of type `IntBits` instead of an `int` and apply the square bracket notation, as shown in the next example:

```
int adapted = 0b0_01111110;
IntBits bits = new IntBits(adapted);
bool peek = bits[6]; // retrieve bool at index 6; peek should be true (1)
bits[0] = true; // set the bit at index 0 to true (1)
bits[3] = false; // set the bit at index 3 to false (0)
Console.WriteLine($"{bits}"); // displays 1110111 (0b0_01110111)
```

This syntax is certainly much easier to understand. It directly and succinctly captures the essence of the problem.

Understanding indexer accessors

When you read an indexer, the compiler automatically translates your array-like code into a call to the `get` accessor of that indexer. Consider the following example:

```
bool peek = bits[6];
```

This statement is converted to a call to the get accessor for `bits`, and the `index` argument is set to 6.

Similarly, if you write to an indexer, the compiler automatically translates your array-like code into a call to the set accessor of that indexer, setting the `index` argument to the value enclosed in the square brackets, as illustrated here:

```
bits[3] = true;
```

This statement is converted to a call to the set accessor for `bits` where `index` is 3. As with ordinary properties, the data you're writing to the indexer (in this case, `true`) is made available inside the set accessor by using the `value` keyword. The type of `value` is the same as the type of the indexer itself (in this case, `bool`).

It's also possible to use an indexer in a combined read/write context. In this case, both the get and set accessors are used. Look at the following statement, which uses the XOR operator (`^`) to invert the value of the bit at index 6 in the `bits` variable:

```
bits[6] ^= true;
```

This code is automatically translated into the following:

```
bits[6] = bits[6] ^ true;
```

This code works because the indexer declares both a get and a set accessor.



Note You can declare an indexer that contains only a get accessor (a read-only indexer) or only a set accessor (a write-only indexer).

Comparing indexers and arrays

When you use an indexer, the syntax is deliberately very array-like. However, there are some important differences between indexers and arrays:

- Indexers can use nonnumeric subscripts, such as a string (as shown in the following example), whereas arrays can use only integer subscripts.

```
public int this [ string name ] { ... } // OK
```

- Indexers can be overloaded (just like methods), whereas arrays cannot.

```
public Name this [ PhoneNumber number ] { ... }  
public PhoneNumber this [ Name name ] { ... }
```

- Indexers cannot be used as `ref` or `out` parameters, whereas array elements can.

```
IntBits bits; // bits contains an indexer  
Method(ref bits[1]); // compile-time error
```

Properties, arrays, and indexers

It's possible for a property to return an array, but remember that arrays are reference types, so exposing an array as a property creates the possibility of accidentally overwriting a lot of data. Look at the following structure that exposes an array property named `Data`:

```
struct Wrapper
{
    private int[] data;
    ...
    public int[] Data
    {
        get => this.data;
        set => this.data = value;
    }
}
```

Now consider the following code that uses this property:

```
Wrapper wrap = new Wrapper();
...
int[] myData = wrap.Data;
myData[0]++;
myData[1]++;
```

This sequence looks pretty innocuous. However, because arrays are reference types, the variable `myData` refers to the same object as the private `data` variable in the `Wrapper` structure. Any changes you make to elements in `myData` are made to the `data` array; the expression `myData[0]++` has the very same effect as `data[0]++`.

If this is not your intention, you should use the `Clone` method in the `get` and `set` accessors of the `Data` property to return a copy of the `data` array or make a copy of the value being set, as shown in the following code. (Chapter 8, “Understanding values and references,” discusses the `Clone` method for copying arrays.) Notice that the `Clone` method returns an object, which you must cast to an integer array.

```
struct Wrapper
{
    private int[] data;
    ...
    public int[] Data
    {
        get { return this.data.Clone() as int[]; }
        set { this.data = value.Clone() as int[]; }
    }
}
```

This approach can become very messy and expensive in terms of memory use. Indexers provide a natural solution to this problem. That is, don't expose the entire array as a property; just make its individual elements available through an indexer:

```
struct Wrapper
{
    private int[] data;
    ...
    public int this [int i]
    {
        get => this.data[i];
        set => this.data[i] = value;
    }
}
```

The following code uses the indexer in a similar manner to the property shown earlier:

```
Wrapper wrap = new Wrapper();
...
int[] myData = new int[2];
myData[0] = wrap[0];
myData[1] = wrap[1];
myData[0]++;
myData[1]++;
```

This time, incrementing the values in the `myData` array has no effect on the original array in the `Wrapper` object. If you really want to modify the data in the `Wrapper` object, you must write statements such as this:

```
wrap[0]++;
```

This is much clearer and safer!

Indexers in interfaces

You can declare indexers in an interface. To do this, specify the `get` keyword, the `set` keyword, or both, but replace the body of the `get` or `set` accessor with a semicolon. Any class or structure that implements the interface must implement the indexer accessors declared in the interface, as demonstrated here:

```
interface IRawInt
{
    bool this [ int index ] { get; set; }
}

struct RawInt : IRawInt
{
    ...
    public bool this [ int index ]
    {
```



```

        get { ... }
        set { ... }
    }
    ...
}

```

If you implement the `indexer` in a class, you can declare the indexer implementations as `virtual`. This allows further derived classes to override the `get` and `set` accessors, such as in the following:

```

class RawInt : IRawInt
{
    ...
    public virtual bool this [ int index ]
    {
        get { ... }
        set { ... }
    }
    ...
}

```

You can also choose to implement an indexer by using the explicit interface implementation syntax covered in Chapter 13, “Creating interfaces and defining abstract classes.” An explicit implementation of an indexer is `nonpublic` and `nonvirtual` (and so cannot be overridden), as shown in this example:

```

struct RawInt : IRawInt
{
    ...
    bool IRawInt.this [ int index ]
    {
        get { ... }
        set { ... }
    }
    ...
}

```

Using indexers in a Windows application

In the following exercise, you’ll examine a simple phone book application and complete its implementation. You’ll write two indexers in the `PhoneBook` class: one that accepts a `Name` parameter and returns a `PhoneNumber` object, and another that accepts a `PhoneNumber` parameter and returns a `Name` object. (The `Name` and `PhoneNumber` structures have already been written.) You’ll also need to call these indexers from the correct places in the program.

To familiarize yourself with the application

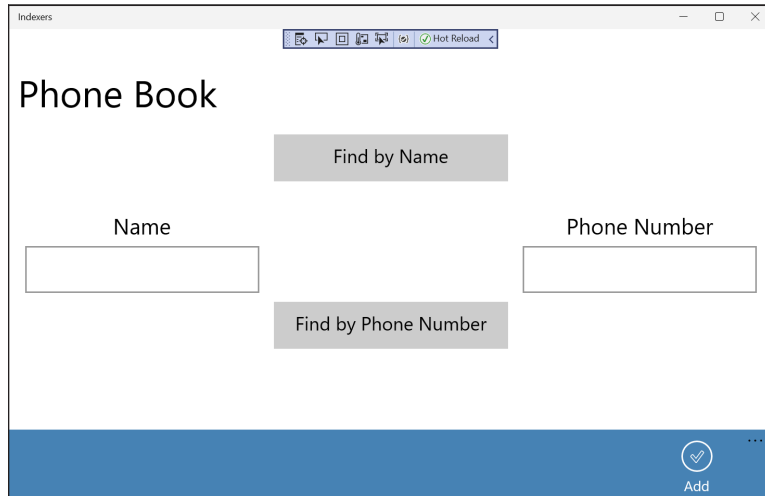
1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **Indexers** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 16\Indexers** folder in your **Documents** folder.

With this graphical application, a user can search for the telephone number for a contact and also find the name of a contact that matches a given telephone number.

3. On the **Debug** menu, select **Start Debugging**.

The project builds and runs. A form appears, displaying two empty boxes labeled Name and Phone Number. The form initially displays two buttons: one to find a phone number when given a name, and one to find a name when given a phone number.

Expanding the command bar at the bottom of the form reveals an additional Add button that will add a name/phone number pair to a list of names and phone numbers held by the application. All buttons (including the Add button in the command bar) currently do nothing. The application looks like this:



Your task is to complete the application so that the buttons work.

4. Return to Visual Studio 2022 and stop debugging.
5. Display the Name.cs file for the Indexers project in the Code and Text Editor window. Then examine the Name structure. Its purpose is to act as a holder for names.

The name is provided as a string to the constructor. The name can be retrieved by using the read-only `string` property named `Text`. (The `Equals` and `GetHashCode` methods are used for comparing Name objects when searching through an array of Name values. You can ignore them for now.)

6. Display the PhoneNumber.cs file in the Code and Text Editor window, and examine the `PhoneNumber` structure. It's similar to the Name structure.
7. Display the PhoneBook.cs file in the Code and Text Editor window and examine the `PhoneBook` class.

This class contains two private arrays: an array of Name values called `names`, and an array of `PhoneNumber` values called `phoneNumbers`. The `PhoneBook` class also contains an `Add` method that adds a phone number and name to the phone book. This method is called when the user selects the Add button on the form.

The `enlargeIfFull` method is called by `Add` to check whether the arrays are full when the user adds another entry. This method creates two new, bigger arrays, copies the contents of the existing arrays to them, and then discards the old arrays.

The `Add` method is deliberately kept simple and does not check whether a name or phone number has already been added to the phone book.

The `PhoneBook` class does not currently provide any functionality with which a user can find a name or telephone number; you'll add two indexers to provide this capability in the next exercise.

To write the indexers

1. In the `PhoneBook.cs` file, delete the comment `// TODO: write 1st indexer here` and replace it with a public read-only indexer for the `PhoneBook` class, as shown in bold in the code that follows. The indexer should return a `Name` object and take a `PhoneNumber` item as its index. Leave the body of the `get` accessor blank.

The indexer should look like this:

```
sealed class PhoneBook
{
    ...
    public Name this[PhoneNumber number]
    {
        get
        {
        }
    }
    ...
}
```

2. Implement the `get` accessor as shown in bold in the code that follows:

```
sealed class PhoneBook
{
    ...
    public Name this [PhoneNumber number]
    {
        get
        {
            int i = Array.IndexOf(this.phoneNumbers, number);
            if (i != -1)
            {
                return this.names[i];
            }
            else
            {
                return new Name();
            }
        }
    }
    ...
}
```

The purpose of the accessor is to find the name that matches the specified phone number. To do this, you need to call the static `IndexOf` method of the `Array` class. The `IndexOf` method searches through an array and returns the index of the first item in the array that matches the specified value. The first argument to `IndexOf` is the array to search through (`phoneNumbers`). The second argument to `IndexOf` is the item for which you are searching. `IndexOf` returns the integer index of the element if it finds it; otherwise, it returns `-1`. If the indexer finds the phone number, it should return the corresponding name; otherwise, it should return an empty `Name` value. (Note that `Name` is a structure, so the default constructor sets its private name field to `null`.)

3. Remove the comment `// TODO: write 2nd indexer here` and replace it with a second public read-only indexer for the `PhoneBook` class that returns a `PhoneNumber` and accepts a single `Name` parameter. Implement this indexer in the same way as the first one, as shown in bold in the code that follows. (Again, note that `PhoneNumber` is a structure and therefore always has a default constructor.) The second indexer should look like this:

```
sealed class PhoneBook
{
    ...
    public PhoneNumber this [Name name]
    {
        get
        {
            int i = Array.IndexOf(this.names, name);
            if (i != -1)
            {
                return this.phoneNumbers[i];
            }
            else
            {
                return new PhoneNumber();
            }
        }
    }
    ...
}
```

Notice that these overloaded indexers can coexist because the values that they index are of different types, which means that their signatures are different. If the `Name` and `PhoneNumber` structures were replaced by simple strings (which they wrap), the overloads would have the same signature, and the class would not compile.

4. On the **Build** menu, select **Build Solution**, correct any syntax errors, and then rebuild the solution if necessary.

To call the indexers

1. Display the `MainPage.xaml.cs` file in the Code and Text Editor window and then locate the `findByNameClick` method.

This method is called when the Find By Name button is selected. This method is currently empty.

2. Replace the `// TODO: comment` with the code shown in bold in the following example:

```
private void findByNameClick(object sender, RoutedEventArgs e)
{
    string text = name.Text;
    if (!String.IsNullOrEmpty(text))
    {
        Name personsName = new Name(text);
        PhoneNumber personsPhoneNumber = this.phoneBook[personsName];
        phoneNumber.Text =
            String.IsNullOrEmpty(personsPhoneNumber.Text) ?
            "Not Found" : personsPhoneNumber.Text;
    }
}
```

This code performs the following tasks:

- It reads the value of the `Text` property from the name box on the form. This is a string containing the contact name that the user has typed in.
- If the string is not empty, the code searches for the phone number corresponding to that name in the `PhoneBook` object by using the indexer. (Notice that the `MainPage` class contains a private `PhoneBook` field named `phoneBook`.) It constructs a `Name` object from the string and passes it as the parameter to the `PhoneBook` indexer.
- If the `Text` property of the `PhoneNumber` structure returned by the indexer is not `null` or empty, the code writes the value of this property to the `phoneNumber` box on the form; otherwise, it displays the text `Not Found`.

Other than the statement that accesses the indexer, there are two further points of interest in this code:

- The static `String` method `IsNullOrEmpty` is used to determine whether a string is empty or contains a `null` value. This is the preferred method for testing whether a string contains a value. It returns `true` if the string contains a `null` value or is an empty string; otherwise, it returns `false`.
 - The `?:` operator is used by the statement that populates the `Text` property of the `phoneNumber` box on the form. Remember that this operator acts like an inline `if...else` statement for an expression. In the preceding code, if the expression `String.IsNullOrEmpty(personsPhoneNumber.Text)` is `true`, no matching entry was found in the phone book and the text `Not Found` is displayed on the form; otherwise, the value held in the `Text` property of the `personsPhoneNumber` variable is displayed.
3. Locate the `findByPhoneNumberClick` method in the `MainPage.xaml.cs` file. It's after the `findByNameClick` method.

The `findByPhoneNumberClick` method is called when the `Find By Phone Number` button is selected. This method is currently empty apart from a `// TODO: comment`.

4. To implement the `findByPhoneNumberClick`, enter the following code shown in bold:

```
private void findByPhoneNumberClick(object sender, RoutedEventArgs e)
{
    string text = phoneNumber.Text;
    if (!String.IsNullOrEmpty(text))
    {
        PhoneNumber personsPhoneNumber = new PhoneNumber(text);
        Name personsName = this.phoneBook[personsPhoneNumber];
        name.Text = String.IsNullOrEmpty(personsName.Text) ?
            "Not Found" : personsName.Text;
    }
}
```

This code performs the following tasks:

- It reads the value of the `Text` property from the `phoneNumber` box on the form. This is a string containing the phone number that the user has typed.
- If the string is not empty, it uses the indexer to search for the name corresponding to that phone number in the `PhoneBook` object.
- It writes the `Text` property of the `Name` structure returned by the indexer to the `name` box on the form.

5. On the **Build** menu, select **Build Solution**, and then correct any errors that occur.

To test the application

1. On the **Debug** menu, select **Start Debugging**.
2. Type your name and phone number in the appropriate boxes, and then expand the command bar and select **Add**. (You can expand the command bar by selecting the ellipsis.)

When you select the **Add** button, the `Add` method stores the information in the phone book and clears the boxes so that they are ready to perform a search.

3. Repeat step 2 several times with some different names and phone numbers so that the phone book contains a selection of entries. Note that the application performs no checking of the names and telephone numbers that you enter, and you can input the same name and telephone number more than once. For the purposes of this demonstration, to avoid confusion, be sure that you provide different names and telephone numbers.
4. Type a name that you used in step 3 into the **Name** box, and then select **Find By Name**.

The phone number you added for the contact whose name you entered in step 3 is retrieved from the phone book and is displayed in the `Phone Number` box.
5. Type a phone number for a different contact in the **Phone Number** box, and then select **Find By Phone Number**.

The contact name is retrieved from the phone book and is displayed in the `Name` box.

6. Type a name that you did not enter in the phone book into the **Name** box, and then select **Find By Name**.

This time, the Phone Number box displays the message Not Found.
7. Close the form and return to Visual Studio 2022.

Summary

In this chapter, you saw how to use indexers to provide array-like access to data in a class. You learned how to create indexers that can take an index and return the corresponding value by using logic defined by the `get` accessor, and you saw how to use the `set` accessor with an index to populate a value in an indexer.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 17, “Introducing generics.”
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

| To | Do this |
|--|--|
| Specify an integer value using binary or hexadecimal notation | Use the <code>0b0</code> (for binary values) or <code>0x0</code> (for hexadecimal values) prefixes. Include <code>_</code> separators to make values easier to read. For example:
<pre>uint moreBinData = 0b0_11110000_01011010_11001100_00001111;
uint moreHexData = 0x0_F0_5A_CC_0F;</pre> |
| Display an integer value as its binary or hexadecimal representation | Use the <code>Convert.ToString</code> method, and specify 2 (for binary) or 16 (for hexadecimal) as the number base. For example:
<pre>uint moreHexData = 0x0_F0_5A_CC_0F;
Console.WriteLine(\$"{Convert.ToString(moreHexData, 2)}");
// displays 11110000010110101100110000001111</pre> |
| Create an indexer for a class or structure | Declare the type of the indexer, followed by the keyword <code>this</code> , and then the indexer arguments in square brackets. The body of the indexer can contain a <code>get</code> and/or <code>set</code> accessor. For example:
<pre>struct RawInt
{
 ...
 public bool this [int index]
 {
 get { ... }
 set { ... }
 }
 ...
}</pre> |
| Define an indexer in an interface | Define an indexer with the <code>get</code> and/or <code>set</code> keywords. For example:
<pre>interface IRawInt
{
 bool this [int index] { get; set; }
}</pre> |

| | |
|---|---|
| Implement an interface indexer in a class or structure | <p>In the class or structure that implements the interface, define the indexer and implement the accessors. For example:</p> <pre> struct RawInt : IRawInt { ... public bool this [int index] { get { ... } set { ... } } ... } </pre> |
| Implement an indexer defined by an interface by using explicit interface implementation in a class or structure | <p>In the class or structure that implements the interface, specify the interface but do not specify the indexer accessibility. For example:</p> <pre> struct RawInt : IRawInt { ... bool IRawInt.this [int index] { get { ... } set { ... } } ... } </pre> |

Introducing generics

After completing this chapter, you will be able to:

- Explain the purpose of generics.
- Define a type-safe class by using generics and create instances of a generic class based on types specified as type parameters.
- Define a generic method that implements an algorithm independent of the type of data on which it operates.
- Create and implement a generic interface.

Chapter 8, “Understanding values and references,” showed you how to use the `object` type to refer to an instance of any class. You can use the `object` type to store a value of any type, and you can define parameters by using the `object` type when you need to pass values of any type into a method. A method can also return values of any type by specifying `object` as the return type. Although this practice is very flexible, it puts the onus on the programmer to remember what sort of data is actually being used. This can lead to runtime errors if the programmer makes a mistake. In this chapter, you’ll learn about generics, a feature that has been designed to help you prevent this kind of mistake.

The problem: Issues with the object type

To understand generics, it’s worth looking in detail at the problem they are designed to solve.

Suppose that you need to model a first-in, first-out structure such as a queue. You could create a class such as the following:

```
class Queue
{
    private const int DEFAULTQUEUESIZE = 100;
    private int[] data;
    private int head = 0, tail = 0;
    private int numElements = 0;

    public Queue()
    {
        this.data = new int[DEFAULTQUEUESIZE];
    }
}
```

```

public Queue(int size)
{
    if (size > 0)
    {
        this.data = new int[size];
    }
    else
    {
        throw new ArgumentOutOfRangeException("size", "Must be greater than zero");
    }
}

public void Enqueue(int item)
{
    if (this.numElements == this.data.Length)
    {
        throw new Exception("Queue full");
    }
    this.data[this.head] = item;
    this.head++;
    this.head %= this.data.Length;
    this.numElements++;
}

public int Dequeue()
{
    if (this.numElements == 0)
    {
        throw new Exception("Queue empty");
    }
    int queueItem = this.data[this.tail];
    this.tail++;
    this.tail %= this.data.Length;
    this.numElements--;
    return queueItem;
}
}

```

This class uses an array to provide a circular buffer for holding the data. The size of this array is specified by the constructor. An application uses the `Enqueue` method to add an item to the queue and the `Dequeue` method to pull an item from the queue. The private `head` and `tail` fields keep track of where to insert an item into the array and where to retrieve an item from the array. The `numElements` field indicates how many items are in the array. The `Enqueue` and `Dequeue` methods use these fields to determine where to store or retrieve an item and perform some rudimentary error checking.

An application can create a `Queue` object and call these methods, as shown in the code example that follows. Notice that the items are dequeued in the same order in which they are enqueued:

```

Queue queue = new Queue(); // Create a new Queue
queue.Enqueue(100);
queue.Enqueue(-25);
queue.Enqueue(33);
Console.WriteLine($"{queue.Dequeue()}"); // Displays 100
Console.WriteLine($"{queue.Dequeue()}"); // Displays -25
Console.WriteLine($"{queue.Dequeue()}"); // Displays 33

```

Now, the `Queue` class works well for queues of `ints`, but what if you want to create queues of `strings`, or `floats`, or even queues of more complex types such as `Circle` (see Chapter 7, “Creating and managing classes and objects”), or `Horse` or `Whale` (see Chapter 12, “Working with inheritance”)? The problem is that the way in which the `Queue` class is implemented restricts it to items of type `int`, and if you try to enqueue a `Horse` object, you’ll get a compile-time error.

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse); // Compile-time error: Cannot convert from Horse to int
```

One way around this restriction is to specify that the array in the `Queue` class contains items of type `object`, update the constructors, and modify the `Enqueue` and `Dequeue` methods to take an `object` parameter and return an `object`, as in the following:

```
class Queue
{
    ...
    private object[] data;
    ...
    public Queue()
    {
        this.data = new object[DEFAULTQUEUESIZE];
    }

    public Queue(int size)
    {
        ...
        this.data = new object[size];
        ...
    }

    public void Enqueue(object item)
    {
        ...
    }

    public object Dequeue()
    {
        ...
        object queueItem = this.data[this.tail];
        ...
        return queueItem;
    }
}
```

Remember that you can use the `object` type to refer to a value or variable of any type. All reference types automatically inherit (either directly or indirectly) from the `System.Object` class in Microsoft .NET. (In C#, `object` is an alias for `System.Object`.) Now, because the `Enqueue` and `Dequeue` methods manipulate objects, you can operate on queues of `Circle` objects, `Horse` objects, `Whale` objects, or objects based on any of the other classes that you’ve seen in this book. However, you must cast the

value returned by the `Dequeue` method to the appropriate type because the compiler will not perform the conversion from the object type automatically.

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse); // Now legal - Horse is an object
...
Horse dequeuedHorse = (Horse)queue.Dequeue(); // Need to cast object back to a Horse
```

If you don't cast the returned value, you'll get the compiler error `Cannot implicitly convert type 'object' to 'Horse.'`

This requirement to perform an explicit cast degenerates much of the flexibility afforded by the object type. Furthermore, it's very easy to write code such as this:

```
Queue queue = new Queue();
Horse myHorse = new Horse();
queue.Enqueue(myHorse);
...
Circle myCircle = (Circle)queue.Dequeue(); // runtime error
```

Although this code will compile, it is not valid and throws a `System.InvalidCastException` exception at runtime. The error is caused by trying to store a reference to a `Horse` object in a `Circle` variable when it is dequeued, and the two types are not compatible. This error isn't spotted until runtime because the compiler doesn't have enough information to perform this check at compile time. The real type of the object being dequeued becomes apparent only when the code runs.

Another disadvantage of using the object approach to create generalized classes and methods is that it can consume additional memory and processor time if the runtime needs to convert an object to a value type and back again. Consider the following piece of code that manipulates a queue of `int` values:

```
Queue queue = new Queue();
int myInt = 99;
queue.Enqueue(myInt); // box the int to an object
...
myInt = (int)queue.Dequeue(); // unbox the object to an int
```

The `Queue` data type expects the items it holds to be objects, and object is a reference type. Enqueueing a value type, such as an `int`, requires it to be boxed to convert it to a reference type. Similarly, dequeuing into an `int` requires the item to be unboxed to convert it back to a value type. (See the sections "Boxing" and "Unboxing" in Chapter 8 for more details.) Although boxing and unboxing happen transparently, they add performance overhead because they involve dynamic memory allocations. This overhead is small for each item, but it adds up when a program creates queues of large numbers of value types.

The generics solution

C# provides generics to remove the need for casting, improve type safety, reduce the amount of boxing required, and make it easier to create generalized classes and methods. Generic classes and methods accept *type parameters*, which specify the types of objects on which they operate. In C#, you indicate that a class is generic by providing a type parameter in angle brackets, like this:

```
class Queue<T>
{
    ...
}
```

The T in this example acts as a placeholder for a real type at compile time. When you write code to instantiate a generic Queue class, you provide the type that should be substituted for T (Circle, Horse, int, and so on). When you define the fields and methods in the class, you use this same placeholder to indicate the type of these items, like this:

```
class Queue<T>
{
    ...
    private T[] data; // array is of type 'T' where 'T' is the type parameter
    ...
    public Queue()
    {
        this.data = new T[DEFAULTQUEUESIZE]; // use 'T' as the data type
    }

    public Queue(int size)
    {
        ...
        this.data = new T[size];
        ...
    }

    public void Enqueue(T item) // use 'T' as the type of the method parameter
    {
        ...
    }

    public T Dequeue() // use 'T' as the type of the return value
    {
        ...
        T queueItem = this.data[this.tail]; // the data in the array is of type 'T'
        ...
        return queueItem;
    }
}
```

The type parameter T can be any legal C# identifier, although the lone character T is commonly used. It's replaced with the type you specify when you create a Queue object. The following examples create a Queue of ints and a Queue of Horses:

```
Queue<int> intQueue = new Queue<int>();
Queue<Horse> horseQueue = new Queue<Horse>();
```

Additionally, the compiler now has enough information to perform strict type-checking when you build the application. You no longer need to cast data when you call the Dequeue method, and the compiler can trap any type mismatch errors early:

```
intQueue.Enqueue(99);
int myInt = intQueue.Dequeue(); // no casting necessary
Horse myHorse = intQueue.Dequeue(); // compiler error: cannot implicitly convert type 'int'
to 'Horse'
```

You should be aware that this substitution of T for a specified type is not simply a textual replacement mechanism. Instead, the compiler performs a complete semantic substitution so that you can specify any valid type for T. Here are more examples:

```
struct Person
{
    ...
}
...
Queue<int> intQueue = new Queue<int>();
Queue<Person> personQueue = new Queue<Person>();
```

The first example creates a queue of integers, whereas the second example creates a queue of Person values. The compiler also generates the versions of the Enqueue and Dequeue methods for each queue. For the intQueue queue, these methods look like this:

```
public void Enqueue(int item);
public int Dequeue();
```

For the personQueue queue, these methods look like this:

```
public void Enqueue(Person item);
public Person Dequeue();
```

Contrast these definitions with those of the object-based version of the Queue class shown in the preceding section. In the methods derived from the generic class, the item parameter to Enqueue is passed as a value type that does not require boxing. Similarly, the value returned by Dequeue is also a value type that doesn't need to be unboxed. A similar set of methods is generated for the other two queues.



Note The `System.Collections.Generic` namespace in the .NET class library provides an implementation to the Queue class that operates similarly to the class just described. This namespace also includes several other collection classes. These are described in more detail in Chapter 18, “Using collections.”

The type parameter doesn't have to be a simple class or value type. For example, you can create a queue of queues of integers (if you ever find it necessary), like this:

```
Queue<Queue<int>> queueQueue = new Queue<Queue<int>>();
```

A generic class can have multiple type parameters. For example, the generic Dictionary class defined in the `System.Collections.Generic` namespace in the .NET class library expects two type parameters: one type for keys, and another for the values. (This class is described in more detail in Chapter 18.)



Note You can also define generic structures and interfaces by using the same type-parameter syntax as for generic classes.

Generics vs. generalized classes

A generic class that uses type parameters is different from a *generalized* class designed to take parameters that can be cast to different types. For example, the object-based version of the `Queue` class shown earlier is a generalized class. There is a single implementation of this class, and its methods take `object` parameters and return `object` types. You can use this class with `ints`, `strings`, and many other types, but in each case, you're using instances of the same class, and you must cast the data you're using to and from the `object` type.

Compare this with the `Queue<T>` class. Each time you use this class with a type parameter (such as `Queue<int>` or `Queue<Horse>`), you cause the compiler to generate an entirely new class that happens to have functionality defined by the generic class. This means that `Queue<int>` is a completely different type from `Queue<Horse>`, but they both happen to have the same behavior.

You can think of a generic class as one that defines a template that is then used by the compiler to generate new type-specific classes on demand. The type-specific versions of a generic class (`Queue<int>`, `Queue<Horse>`, and so on) are referred to as *constructed types*, and you should treat them as distinctly different types (albeit types that have a similar set of methods and properties).

Generics and constraints

Occasionally, you'll want to ensure that the type parameter used by a generic class identifies a type that provides certain methods. For example, if you're defining a `PrintableCollection` class, you might want to ensure that all objects stored in the class have a `Print` method. You can specify this condition by using a *constraint*.

By using a constraint, you can limit the type parameters of a generic class to those that implement a particular set of interfaces and therefore provide the methods defined by those interfaces. For example, if the `IPrintable` interface defined the `Print` method, you could create the `PrintableCollection` class like this:

```
public class PrintableCollection<T> where T : IPrintable
```

When you build this class with a type parameter, the compiler checks to ensure that the type used for `T` actually implements the `IPrintable` interface; if it doesn't, it stops with a compilation error.

You can also force the type parameter to be a `struct` value type rather than a class by using the `struct` constraint:

```
public class StructCollection<T> where T : struct
```

The `class` constraint is similar, except that it ensures that the type parameter is a reference type (a class).

Creating a generic class

The `System.Collections.Generic` namespace in the .NET class library contains some generic classes readily available for you. You can also define your own generic classes, which is what you'll do in this section. Before you do this, let's cover a bit of background theory.

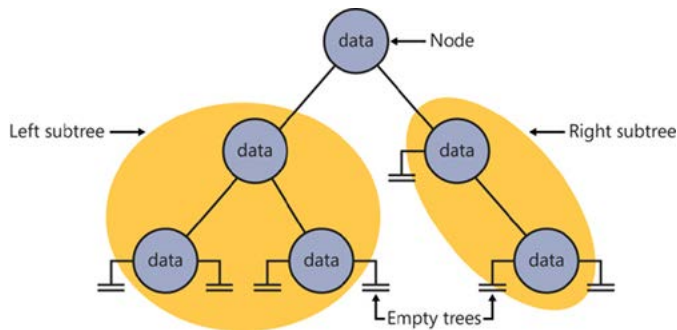
The theory of binary trees

In the following exercises, you'll define and use a class that represents a binary tree. A *binary tree* is a data structure that you can use for a variety of operations, including sorting and searching through data very quickly.



Note Volumes have been written on the minutiae of binary trees, but it is not the purpose of this book to cover this topic in detail. Instead, you'll look at just the pertinent facts. If you're interested in learning more, consult *The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd Edition* by Donald E. Knuth (Addison-Wesley Professional, 1998). Despite its age, this is the recognized, seminal work on sort and search algorithms.

A binary tree is a recursive (self-referencing) data structure that can be empty or contain three elements: a datum, which is typically referred to as the *node*, and two subtrees, which are themselves binary trees. The two subtrees are conventionally called the *left subtree* and the *right subtree* because they're typically depicted to the left and right of the node, respectively. Each left subtree or right subtree is either empty or contains a node and other subtrees. In theory, the whole structure can continue ad infinitum. The following image shows the structure of a small binary tree:



The real power of binary trees becomes evident when you use them for sorting data. If you start with an unordered sequence of objects of the same type, you can construct an ordered binary tree and then walk through the tree to visit each node in an ordered sequence. The algorithm for inserting an item I into an ordered binary tree B is shown here:

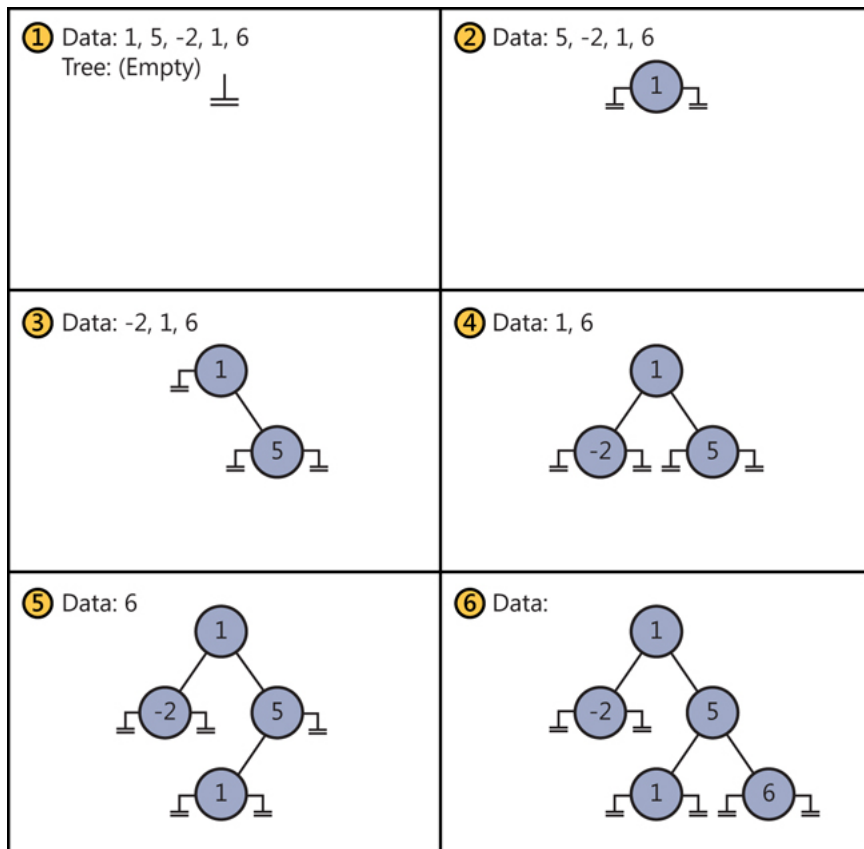
```
If the tree, B, is empty
Then
    Construct a new tree B with the new item I as the node, and empty left and right subtrees
Else
    Examine the value of the current node, N, of the tree, B
    If the value of N is greater than that of the new item, I
    Then
        If the left subtree of B is empty
        Then
            Construct a new left subtree of B with the item I as the node, and empty left and right
            subtrees
        Else
            Insert I into the left subtree of B
        End If
    Else
        If the right subtree of B is empty
        Then
            Construct a new right subtree of B with the item I as the node, and empty left and right
            subtrees
        Else
            Insert I into the right subtree of B
        End If
    End If
End If
```

Notice that this algorithm is recursive, calling itself to insert the item into the left or right subtree depending on how the value of the item compares with the current node in the tree.



Note The definition of the expression *greater than* depends on the type of data in the item and node. For numeric data, greater than can be a simple arithmetic comparison; for text data, it can be a string comparison. However, you must give other forms of data their own means of comparing values. You'll learn more about this when you implement a binary tree in the next section.

If you start with an empty binary tree and an unordered sequence of objects, you can iterate through the unordered sequence, inserting each object into the binary tree by using this algorithm, resulting in an ordered tree. The next image shows the steps in the process for constructing a tree from a set of five integers.



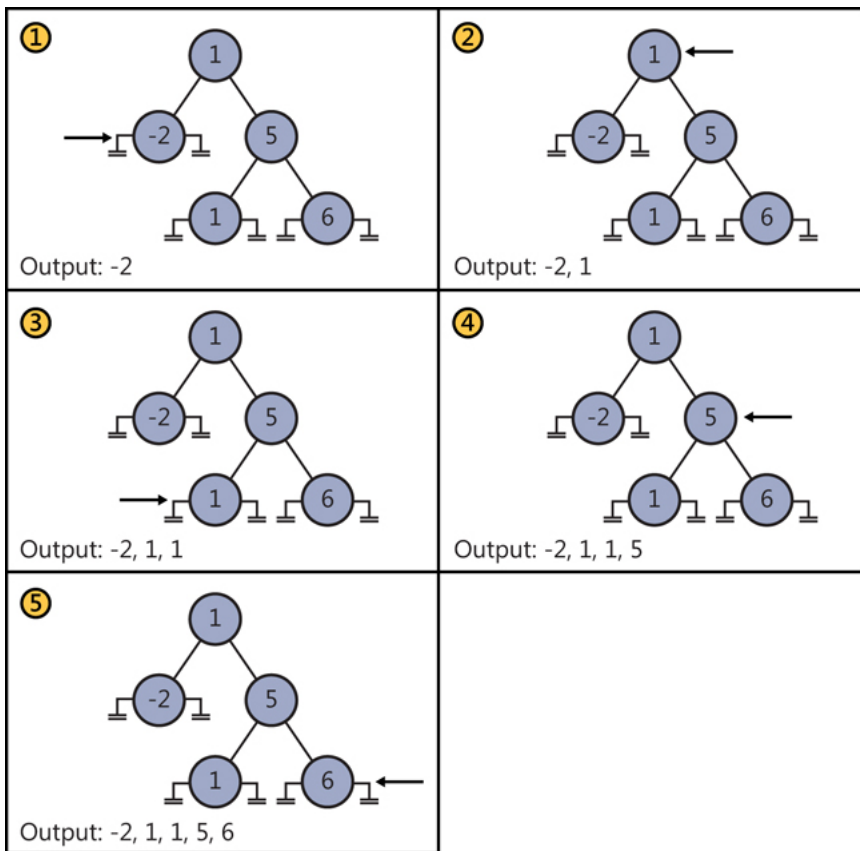
After you've built an ordered binary tree, you can display its contents in sequence by visiting each node in turn and printing the value found. The algorithm for achieving this task is also recursive:

```

If the left subtree is not empty
Then
    Display the contents of the left subtree
End If
Display the value of the node
If the right subtree is not empty
Then
    Display the contents of the right subtree
End If

```

The following image shows the steps in the process for outputting the tree. Notice that the integers are now displayed in ascending order.



Building a binary tree class by using generics

In the following exercise, you'll use generics to define a binary tree class capable of holding almost any type of data. The only restriction is that the data type must provide a means of comparing values between different instances.

You might find the binary tree class useful in many different applications. Therefore, you'll implement it as a class library rather than as an application in its own right. You can then use this class elsewhere without having to copy the source code and recompile it. A *class library* is a set of compiled classes (and other types such as structures and delegates) stored in an assembly. An *assembly* is a file that usually has the .dll suffix. Other projects and applications can use the items in a class library by adding a reference to its assembly and then bringing its namespaces into scope by employing using directives. You'll do this when you test the binary tree class.

The System.IComparable and System.IComparable<T> interfaces

The algorithm for inserting a node into a binary tree requires you to compare the value of the node that you're inserting with nodes already in the tree. If you're using a numeric type, such as `int`, you can use the `<`, `>`, and `==` operators. However, if you're using some other type, such as `Mammal` or `Circle` described in earlier chapters, how do you compare objects?

If you need to create a class that requires you to be able to compare values according to some natural (or possibly unnatural) ordering, you should implement the `IComparable` interface. This interface contains a method called `CompareTo`, which takes a single parameter specifying the object to be compared with the current instance and returns an integer that indicates the result of the comparison, as summarized by the following table:

| Value | Meaning |
|----------------|--|
| Less than 0 | The current instance is less than the value of the parameter. |
| 0 | The current instance is equal to the value of the parameter. |
| Greater than 0 | The current instance is greater than the value of the parameter. |

As an example, consider the `Circle` class described in Chapter 7. Let's look at it again here:

```
class Circle
{
    public Circle(int initialRadius)
    {
        radius = initialRadius;
    }

    public double Area()
    {
        return Math.PI * radius * radius;
    }

    private double radius;
}
```

You can make the `Circle` class “comparable” by implementing the `System.IComparable` interface and providing the `CompareTo` method. In this example, the `CompareTo` method compares `Circle` objects based on their areas. A circle with a larger area is considered to be greater than a circle with a smaller area.

```
class Circle : System.IComparable
{
    ...
    public int CompareTo(object obj)
    {
        Circle circObj = (Circle)obj; // cast the parameter to its real type
        if (this.Area() == circObj.Area())
            return 0;
    }
}
```

```

        if (this.Area() > circObj.Area())
            return 1;
        return -1;
    }
}

```

If you examine the `System.IComparable` interface, you'll see that its parameter is defined as an object. However, this approach is not type safe. To understand why, consider what happens if you try to pass something that is not a `Circle` object to the `CompareTo` method. The `System.IComparable` interface requires the use of a cast to access the `Area` method. If the parameter is not a `Circle` but some other type of object, this cast will fail. However, the `System` namespace also defines the generic `IComparable<T>` interface, which contains the following method:

```
int CompareTo(T other);
```

Notice that this method takes a type parameter (`T`) rather than an object, and therefore it's much safer than the nongeneric version of the interface. The following code shows how you can implement this interface in the `Circle` class:

```

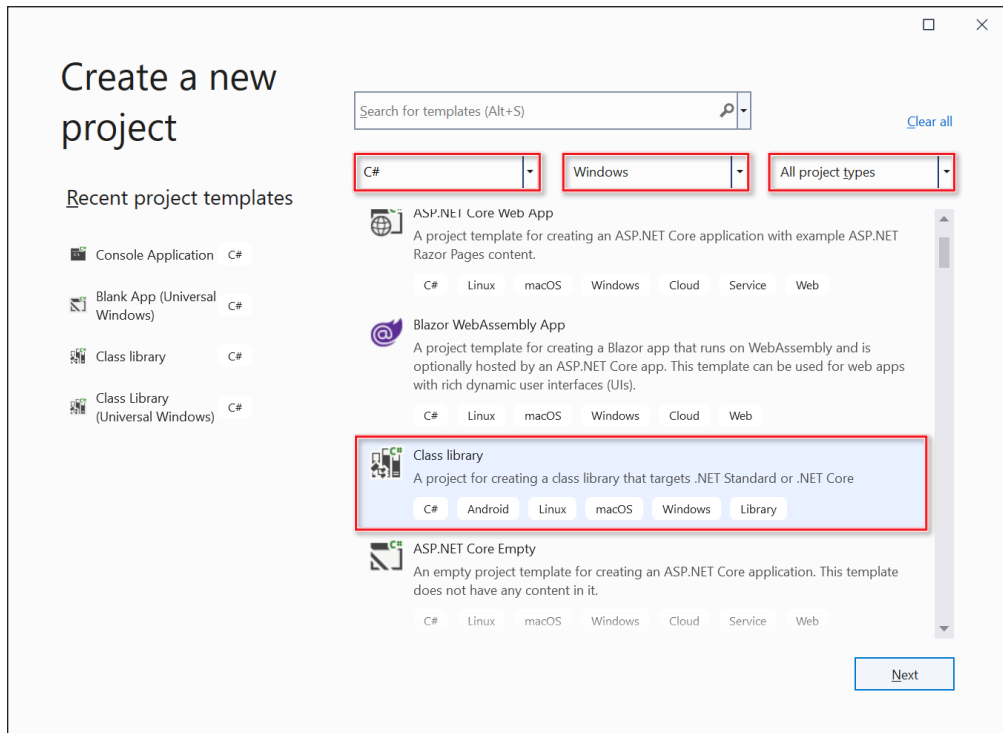
class Circle : System.IComparable<Circle>
{
    ...
    public int CompareTo(Circle other)
    {
        if (this.Area() == other.Area())
            return 0;
        if (this.Area() > other.Area())
            return 1;
        return -1;
    }
}

```

The parameter for the `CompareTo` method must match the type specified in the interface `IComparable<Circle>`. In general, it's preferable to implement the `System.IComparable<T>` interface rather than the `System.IComparable` interface. You can also implement both, and many types in .NET do.

To create the Tree<TItem> class

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. On the **File** menu, point to **New**, and then select **Project**.
3. In the Create a new project dialog, set the **Language** dropdown list box to **C#**, the **Platform** dropdown list box to **Windows**, and the **Project Type** dropdown list box to **All project types**. Select the **Class Library** template, and then select **Next**:



4. In the Configure your new project dialog, in the **Project name** box, type **BinaryTree**. In the **Location** box, specify **\Microsoft Press\VCSBS\Chapter 17** in your **Documents** folder. Then select **Next**.
5. In the Additional information dialog, set the framework to **.NET 6.0**, and then select **Create**.

Using the Class Library template, you can create assemblies that can be reused by different applications. To use a class in a class library in an application, you must first copy the assembly containing the compiled code for the class library to your computer (if you didn't create it yourself) and then add a reference to this assembly.

6. In Solution Explorer, right-click **Class1.cs**, select **Rename**, and change the name of the file to **Tree.cs**. Allow Visual Studio to change the name of the class as well as the name of the file when you're prompted.
7. In the Code and Text Editor window, change the definition of the Tree class to **Tree<TItem>**, as shown in bold in the following code:

```
public class Tree<TItem>
{
}
```

8. In the Code and Text Editor window, modify the definition of the **Tree<TItem>** class to specify that the type parameter **TItem** must denote a type that implements the generic **IComparable<TItem>** interface. The modified definition of the **Tree<TItem>** class should look like this (the changes are highlighted in bold):

```
public class Tree<TItem> where TItem : IComparable<TItem>
{
}
```

9. Add three public automatic properties to the **Tree<TItem>** class: a **TItem** property called **NodeData** and **Tree<TItem>** properties called **LeftTree** and **RightTree**, as shown in the following code example in bold. Note that the **LeftTree** and **RightTree** properties are both explicitly declared as nullable types.

```
public class Tree<TItem> where TItem : IComparable<TItem>
{
    public TItem NodeData { get; set; }
    public Tree<TItem>? LeftTree { get; set; }
    public Tree<TItem>? RightTree { get; set; }
}
```

10. Add a constructor to the **Tree<TItem>** class that takes a single **TItem** parameter called **nodeValue**. In the constructor, set the **NodeData** property to **nodeValue**, and initialize the **LeftTree** and **RightTree** properties to **null**, as shown in bold in the following code:

```
public class Tree<TItem> where TItem : IComparable<TItem>
{
    ...
    public Tree(TItem nodeValue)
    {
        this.NodeData = nodeValue;
        this.LeftTree = null;
        this.RightTree = null;
    }
}
```



Note The name of the constructor does not include the type parameter; it's called **Tree**, not **Tree<TItem>**.

11. Add a public method called `Insert` to the `Tree<TItem>` class as shown in bold in the code that follows. This method will insert a `TItem` value into the tree (once you've completed it). The method definition should look like this:

```
public class Tree<TItem> where TItem: IComparable<TItem>
{
    ...
    public void Insert(TItem newItem)
    {
    }
}
```

You'll implement the recursive algorithm described earlier for creating an ordered binary tree. The constructor creates the initial node of the tree, so the `Insert` method can assume that the tree is not empty. The code that follows is the part of the algorithm that runs after checking whether the tree is empty. It's reproduced here to help you understand the code you'll write for the `Insert` method in the following steps:

```
...
Examine the value of the node, N, of the tree, B
If the value of N is greater than that of the new item, I
Then
    If the left subtree of B is empty
    Then
        Construct a new left subtree of B with the item I as the node, and empty
        left and right subtrees
    Else
        Insert I into the left subtree of B
    End If
...

```

12. In the `Insert` method, add a statement that declares a local variable of type `TItem`, called `currentNodeValue`. Initialize this variable to the value of the `NodeData` property of the tree, as shown in bold in the following example:

```
public void Insert(TItem newItem)
{
    TItem currentNodeValue = this.NodeData;
}
```

13. Add the `if...else` statement shown in bold in the following code to the `Insert` method after the definition of the `currentNodeValue` variable. This statement uses the `CompareTo` method of the `IComparable<T>` interface to determine whether the value of the current node is greater than that of the new item:

```
public void Insert(TItem newItem)
{
    TItem currentNodeValue = this.NodeData;
    if (currentNodeValue.CompareTo(newItem) > 0)
    {
        // Insert the new item into the left subtree
    }
    else
    {

```

```

        // Insert the new item into the right subtree
    }
}

```

14. In the `if` part of the code, immediately after the comment `// Insert the new item into the left subtree`, add the following statements:

```

if (this.LeftTree is null)
{
    this.LeftTree = new Tree<TItem>(newItem);
}
else
{
    this.LeftTree.Insert(newItem);
}

```

These statements check whether the left subtree is empty. If so, a new tree is created using the new item, and it is attached as the left subtree of the current node; otherwise, the new item is inserted into the existing left subtree by calling the `Insert` method recursively.

15. In the `else` part of the outermost `if...else` statement, immediately after the comment `// Insert the new item into the right subtree`, add the equivalent code that inserts the new node into the right subtree:

```

if (this.RightTree is null)
{
    this.RightTree = new Tree<TItem>(newItem);
}
else
{
    this.RightTree.Insert(newItem);
}

```

16. Add another public method called `WalkTree` to the `Tree<TItem>` class after the `Insert` method. This method walks through the tree, visiting each node in sequence, and generates a string representation of the data that the tree contains. The method definition should look like this:

```

public string WalkTree()
{
}

```

17. Add the statements shown in bold in the code that follows to the `WalkTree` method. These statements implement the algorithm described earlier for traversing a binary tree. As each node is visited, the node value is appended to the string returned by the method:

```

public string WalkTree()
{
    string result = "";

    if (this.LeftTree is not null)
    {
        result = this.LeftTree.WalkTree();
    }

    result += $" {this.NodeData.ToString()} ";
}

```

```

        if (this.RightTree is not null)
        {
            result += this.RightTree.WalkTree();
        }
        return result;
    }
}

```

18. On the **Build** menu, select **Build Solution**. The class should compile cleanly, but correct any errors that are reported and rebuild the solution if necessary.

In the next exercise, you'll test the `Tree<TItem>` class by creating binary trees of integers and strings.

To test the `Tree<TItem>` class

1. In Solution Explorer, right-click the **BinaryTree** solution, point to **Add**, and then select **New Project**.



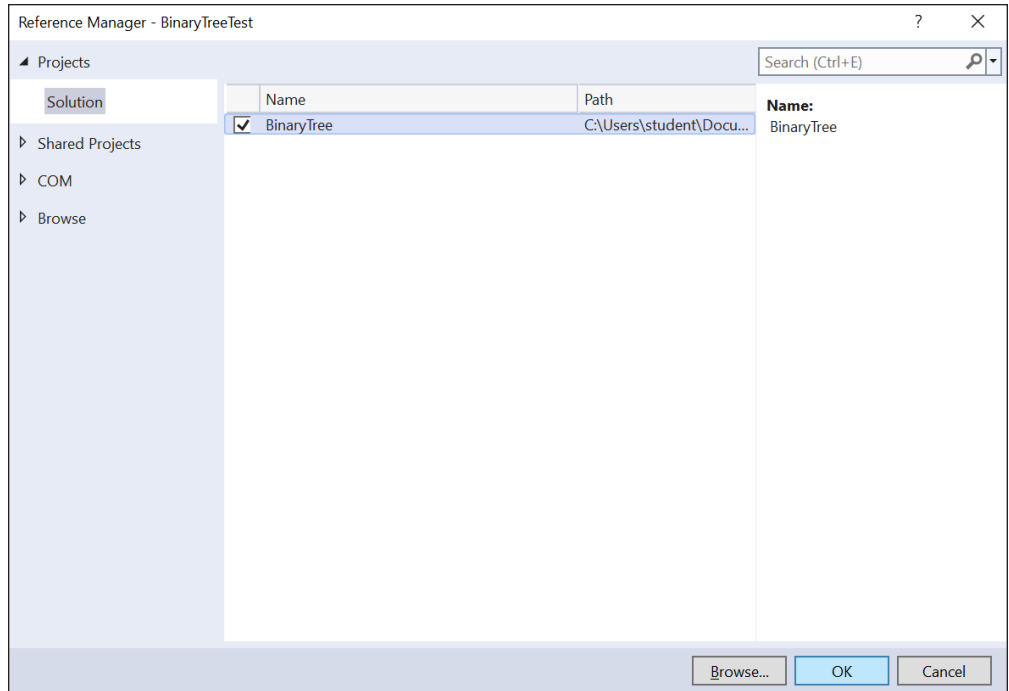
Note Be sure you right-click the *BinaryTree solution* rather than the *BinaryTree project*.

2. Add a new project by using the Console Application template. Give the project the name **BinaryTreeTest**. Set the location to `\Microsoft Press\VCSBS\Chapter 17` in your **Documents** folder, and select the **.NET 6.0** framework.



Note A Visual Studio 2022 solution can contain more than one project. You're using this feature to add a second project to the *BinaryTree* solution for testing the `Tree<TItem>` class.

3. In Solution Explorer, right-click the **BinaryTreeTest** project, and then select **Set As Startup Project**. The *BinaryTreeTest* project is highlighted in Solution Explorer. When you run the application, this is the project that will actually execute.
4. In Solution Explorer, right-click the **BinaryTreeTest** project, point to **Add**, and then select **Project Reference**. The Reference Manager dialog appears. You use this dialog to add a reference to an assembly. This enables you to use the classes and other types implemented by that assembly in your code.
5. In the left pane of the Reference Manager *BinaryTreeTest* dialog, expand **Projects** and then select **Solution**. In the middle pane, select the **BinaryTree** project (be sure to select the checkbox and not simply click or tap on the assembly), and then select **OK**.



This step adds the `BinaryTree` assembly to the list of references for the `BinaryTreeTest` project in Solution Explorer. If you examine the References folder for the `BinaryTreeTest` project in Solution Explorer, you should see the `BinaryTree` assembly listed at the top. You will now be able to create `Tree<TItem>` objects in the `BinaryTreeTest` project.



Note If the class library project is not part of the same solution as the project that uses it, you must add a reference to the assembly (the DLL file) and not to the class library project. You can do this by browsing for the assembly in the Reference Manager dialog. You'll use this technique in the final set of exercises in this chapter.

6. In the Code and Text Editor window displaying the `Program` class in the `Program.cs` file, add the following `using` directive to the list at the top of the class:

```
using BinaryTree;
```

7. Replace the existing statement in the `Main` method with the code shown here in bold:

```
static void Main(string[] args)
{
    Tree<int> tree1 = new Tree<int>(10);
    tree1.Insert(5);
    tree1.Insert(11);
    tree1.Insert(5);
    tree1.Insert(-12);
    tree1.Insert(15);
    tree1.Insert(0);
}
```

```

        tree1.Insert(14);
        tree1.Insert(-8);
        tree1.Insert(10);
        tree1.Insert(8);
        tree1.Insert(8);
        string sortedData = tree1.WalkTree();
        Console.WriteLine($"Sorted data is: {sortedData}");
    }

```

These statements create a new binary tree for holding `ints`. The constructor creates an initial node containing the value 10. The `Insert` statements add nodes to the tree, and the `WalkTree` method generates a string representing the contents of the tree, which should appear sorted in ascending order when this string is displayed.



Note Remember that the `int` keyword in C# is just an alias for the `System.Int32` type. Whenever you declare an `int` variable, you're actually declaring a struct variable of type `System.Int32`. The `System.Int32` type implements the `IComparable` and `IComparable<T>` interfaces, which is why you can create `Tree<int>` objects. Similarly, the `string` keyword is an alias for `System.String`, which also implements `IComparable` and `IComparable<T>`.

8. On the **Build** menu, select **Build Solution**. Correct any errors if necessary.

9. On the **Debug** menu, select **Start Without Debugging**.

Verify that the program runs and displays the values in the following sequence:

```
-12 -8 0 5 5 8 8 10 10 11 14 15
```

10. Press **Enter** to return to Visual Studio 2022.

11. Add the following statements shown in bold to the end of the `Main` method in the `Program` class, after the existing code:

```

static void Main(string[] args)
{
    ...
    Tree<string> tree2 = new Tree<string>("Hello");
    tree2.Insert("World");
    tree2.Insert("How");
    tree2.Insert("Are");
    tree2.Insert("You");
    tree2.Insert("Today");
    tree2.Insert("I");
    tree2.Insert("Hope");
    tree2.Insert("You");
    tree2.Insert("Are");
    tree2.Insert("Feeling");
    tree2.Insert("Well");
    tree2.Insert("!");
    sortedData = tree2.WalkTree();
    Console.WriteLine($"Sorted data is: {sortedData}");
}

```

These statements create another binary tree for holding strings, populate it with some test data, and then print the tree. This time, the data should be sorted alphabetically. The `System.String` class (string is an alias for `System.String`) implements the `Comparable` and `Comparable<string>` interfaces.

12. On the **Build** menu, select **Build Solution**, and verify that the solution compiles. Correct any errors if necessary.
13. On the **Debug** menu, select **Start Without Debugging**.
14. Verify that the program runs and displays the integer values as before, followed by the strings in the following sequence:

! Are Are Feeling Hello Hope How I Today Well World You You
15. Press **Enter** to return to Visual Studio 2022.

Creating a generic method

As well as defining generic classes, you can create generic methods. With a generic method, you can specify the types of the parameters and the return type by using a type parameter like the one used to define a generic class. In this way, you can define generalized methods that are type safe and avoid the overhead of casting (and boxing, in some cases). Generic methods are frequently used in conjunction with generic classes; you need them for methods that take generic types as parameters or that have a return type that's a generic type.

You define generic methods by using the same type parameter syntax you use when you create generic classes. (You can also specify constraints.) For example, the generic `Swap<T>` method in the code that follows swaps the values in its parameters. Because this functionality is useful regardless of the type of data being swapped, it's helpful to define it as a generic method:

```
static void Swap<T>(ref T first, ref T second)
{
    T temp = first;
    first = second;
    second = temp;
}
```

You invoke the method by specifying the appropriate type for its type parameter. The following examples show how to invoke the `Swap<T>` method to swap over two `ints` and two `strings`:

```
int a = 1, b = 2;
Swap<int>(ref a, ref b);
...
string s1 = "Hello", s2 = "World";
Swap<string>(ref s1, ref s2);
```



Note Just as instantiating a generic class with different type parameters causes the compiler to generate different types, each distinct use of the `Swap<T>` method causes the compiler to generate a different version of the method. `Swap<int>` is not the same method as `Swap<string>`—both methods just happen to have been generated from the same generic template, so they exhibit the same behavior, albeit over different types.

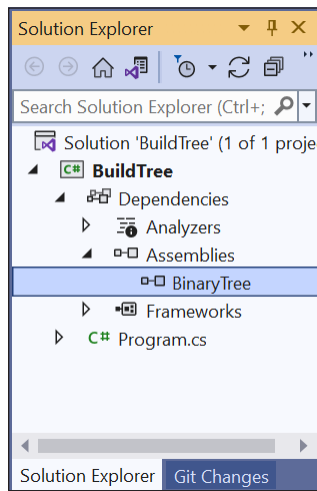
Defining a generic method to build a binary tree

In the previous exercise, you created a generic class for implementing a binary tree. The `Tree<TItem>` class provides the `Insert` method for adding data items to the tree. However, if you want to add a large number of items, repeated calls to the `Insert` method are not very convenient. In the following exercise, you'll define a generic method called `InsertIntoTree` that you can use to insert a list of data items into a tree with a single method call. You'll test this method by using it to insert a list of characters into a tree of characters.

To write the `InsertIntoTree` method

1. In Visual Studio 2022, on the **File** menu, select **Close Solution**. Save the solution if prompted.
2. Create a new project by using the Console Application template. Name the project **BuildTree**. Set the location to **\Microsoft Press\VCSBS\Chapter 17** in your **Documents** folder, and select the **.NET 6.0** framework.
3. On the **Project** menu, select **Add Project Reference**.
4. In the Reference Manager – BuildTree dialog, select the **Browse** button (not the Browse tab in the left pane).
5. In the Select the files to reference dialog, browse to the folder **\Microsoft Press\VCSBS\Chapter 17\BinaryTree\BinaryTree\bin\Debug\net6.0** in your **Documents** folder, select **BinaryTree.dll**, and then select **Add**.
6. In the Reference Manager – BuildTree dialog, verify that the **BinaryTree.dll** assembly is listed and that the checkbox for this assembly is selected, and then select **OK**.

7. In the Solution Explorer window, expand **Dependencies**, and then expand **Assemblies**. Verify that the `BinaryTree` assembly has been added to the list of references for the project.



8. In the Code and Text Editor window displaying the `Program.cs` file, add the following `using` directive to the top of the `Program.cs` file:

```
using BinaryTree;
```

Remember, this namespace contains the `Tree<TItem>` class.

9. After the `Main` method in the `Program` class, add a method named `InsertIntoTree`. This method should be declared as a `static void` method that takes a `Tree<TItem>` parameter and a `params TItem[] data` array of `TItem` elements called `data`. The `tree` parameter should be passed by reference, for reasons that will be described in a later step.

The method definition should look like this:

```
static void InsertIntoTree<TItem>(ref Tree<TItem> tree, params TItem[] data)  
{  
}
```

10. The `TItem` type used for the elements being inserted into the binary tree must implement the `IComparable<TItem>` interface. Modify the definition of the `InsertIntoTree` method and add the `where` clause shown in bold in the following code:

```
static void InsertIntoTree<TItem>(ref Tree<TItem> tree, params TItem[] data)  
    where TItem :IComparable<TItem>  
{  
}
```


11. Add the statements shown in bold in the example that follows to the `InsertIntoTree` method.

These statements iterate through the `params` list, adding each item to the tree by using the `Insert` method. If the value specified by the `tree` parameter is `null` initially, a new `Tree<TItem>` is created; this is why the `tree` parameter is passed by reference.

```
static void InsertIntoTree<TItem>(ref Tree<TItem> tree, params TItem[] data)
    where TItem : IComparable<TItem>
{
    foreach (TItem datum in data)
    {
        if (tree is null)
        {
            tree = new Tree<TItem>(datum);
        }
        else
        {
            tree.Insert(datum);
        }
    }
}
```

To test the `InsertIntoTree` method

1. In the `Main` method of the `Program` class, replace the existing code and add the following statements shown in bold. This code creates a new `Tree` for holding character data, populates it with some sample data by using the `InsertIntoTree` method, and then displays it by using the `WalkTree` method of `Tree`:

```
static void Main(string[] args)
{
    Tree<char> charTree = null;
    InsertIntoTree<char>(ref charTree, 'M', 'X', 'A', 'M', 'Z', 'Z', 'N');
    string sortedData = charTree.WalkTree();
    Console.WriteLine($"Sorted data is: {sortedData}");
}
```

2. On the **Build** menu, select **Build Solution**, verify that the solution compiles, and then correct any errors if necessary.
3. On the **Debug** menu, select **Start Without Debugging**.
4. Verify that the program runs and displays the character values in the following order:
A M M N X Z Z
5. Press **Enter** to return to Visual Studio 2022.

Pointers and memory management revisited

If you're not a C or C++ programmer moving to C#, you can skip this sidebar!

Chapter 8 includes a sidebar describing how you can write C# code that uses pointers to objects directly rather than using type-safe references. The rationale behind this mechanism is to optimize access to data. The disadvantage of this approach is that it's unsafe; you must explicitly mark your code as such to absolve .NET of any problems that might occur from your application stomping blindly over chunks of memory.

In the years since C# and the .NET Framework have been released, Microsoft has added some new types to the .NET library that aim to remove the need for writing unsafe code. These are the generic `Span<T>` and `Memory<T>` types.

The `Span<T>` type represents a contiguous block of memory. It's a little like an array, except that it's a `ref struct` type that's created on the stack. You can use a `Span<T>` object like a fast, managed array. The `Span<T>` type has several useful extension methods that enable you to sort and search the items in memory. Consider the following `struct` definition, used to model a `Person` struct. The struct implements the generic `IComparable<Person>` interface, which provides the `CompareTo` method for comparing one `Person` object against another by using the value in the `Name` field:

```
struct Person : IComparable<Person>
{
    public string Name;

    public int CompareTo(Person other)
    {
        return this.Name.CompareTo(other.Name);
    }
}
```

You can create and populate an *on-stack* array of `Person` structs, like this:

```
var personList = new Span<Person>(new Person[10]);
personList[0] = new Person() { Name = "John" };
personList[1] = new Person() { Name = "Diana" };
...
```

You can then sort and search the data, like this:

```
personList.Sort(); // performs an in-situ sort of the data in the Span<T> object
int item = personList.BinarySearch(new Person() { Name = "Diana" });
```

The value in the `item` variable will be the index of the first `Person` struct found that has the value `Diana` in the `Name` field. The `Sort` and `BinarySearch` methods both use the `CompareTo` method to perform their work. If you have a large amount of data, benchmark tests performed by Microsoft have shown that using a `Span<T>` struct to sort and search data is significantly quicker than using a regular array.

As well as basing a `Span<T>` struct on a specific data type, such as `Person`, you can also use it to create an amorphous blob of safe memory—that is, memory that prevents you from accidentally going beyond the bounds specified when it is allocated. You can do this with the C# `stackalloc` operator. The following example allocates a chunk of memory of 1,000 bytes on the stack:

```
Span<byte> chunk = stackalloc byte[1000];
```

If you needed to perform a fast summation of the byte values held in this block, you could write a method such this:

```
public static int Sum(Span<byte> block)
{
    int sum = 0;
    foreach(byte b in block)
    {
        sum += b;
    }

    return sum;
}
```

This code safely iterates through the block of memory. You can't exceed its bounds without throwing an exception. This approach provides many of the speed advantages of using unsafe code to access memory directly, while obviating many of the disadvantages.

There's also a read-only version of the `Span<T>` type, called `ReadOnlySpan<T>`. You can initialize the data in a `ReadOnlySpan<T>` object when you create it, but thereafter you cannot modify it.

Finally, there are heap-based analogs of `Span<T>` and `ReadOnlySpan<T>`. These types are called `Memory<T>` and `ReadOnlyMemory<T>`. They allocate memory directly from managed heap. For more information about these types, consult the documentation provided with Visual Studio 2022.

Variance and generic interfaces

Chapter 8 demonstrated that you can use the `object` type to hold a value or reference of any other type. For example, the following code is completely legal:

```
string myString = "Hello";
object myObject = myString;
```

Remember that in inheritance terms, the `String` class is derived from the `Object` class, so all strings are objects.

Now consider the following generic interface and class:

```
interface IWrapper<T>
{
    void SetData(T data);
    T GetData();
}

class Wrapper<T> : IWrapper<T>
{
    private T storedData;

    void IWrapper<T>.SetData(T data)
    {
        this.storedData = data;
    }

    T IWrapper<T>.GetData()
    {
        return this.storedData;
    }
}
```

The `Wrapper<T>` class provides a simple wrapper around a specified type. The `IWrapper` interface defines the `SetData` method that the `Wrapper<T>` class implements to store the data and the `GetData` method that the `Wrapper<T>` class implements to retrieve the data. You can create an instance of this class and use it to wrap a string like this:

```
Wrapper<string> stringWrapper = new Wrapper<string>();
IWrapper<string> storedStringWrapper = stringWrapper;
storedStringWrapper.SetData("Hello");
Console.WriteLine($"Stored value is {storedStringWrapper.GetData()}");
```

The code creates an instance of the `Wrapper<string>` type. It references the object through the `IWrapper<string>` interface to call the `SetData` method. (The `Wrapper<T>` type implements its interfaces explicitly, so you must call the methods through an appropriate interface reference.) The code also calls the `GetData` method through the `IWrapper<string>` interface. If you run this code, it outputs the message `Stored value is Hello`.

Look at the following line of code:

```
IWrapper<object> storedObjectWrapper = stringWrapper;
```

This statement is similar to the one that creates the `IWrapper<string>` reference in the previous code example, with the difference being that the type parameter is `object` rather than `string`. Is this code legal? Remember that all strings are objects (you can assign a `string` value to an object reference, as shown earlier), so in theory, this statement looks promising. However, if you try it, the statement will fail to compile with the message `Cannot implicitly convert type 'Wrapper<string>' to 'IWrapper<object>'`.

You can try an explicit cast such as this:

```
IWrapper<object> storedObjectWrapper = (IWrapper<object>)stringWrapper;
```

This code compiles but will fail at runtime with an `InvalidCastException` exception. The problem is that although all strings are objects, the converse is not true. If this statement were allowed, you could write code like this, which ultimately attempts to store a `Circle` object in a `string` field:

```
IWrapper<object> storedObjectWrapper = (IWrapper<object>)stringWrapper;  
Circle myCircle = new Circle();  
storedObjectWrapper.SetData(myCircle);
```

The `IWrapper<T>` interface is said to be *invariant*. You can't assign an `IWrapper<A>` object to a reference of type `IWrapper<B>`, even if type `A` is derived from type `B`. By default, C# implements this restriction to ensure the type safety of your code.

Covariant interfaces

Suppose that you defined the `IStoreWrapper<T>` and `IRetrieveWrapper<T>` interfaces, shown in the following example, in place of `IWrapper<T>` and implemented these interfaces in the `Wrapper<T>` class, like this:

```
interface IStoreWrapper<T>  
{  
    void SetData(T data);  
}  
  
interface IRetrieveWrapper<T>  
{  
    T GetData();  
}  
  
class Wrapper<T> : IStoreWrapper<T>, IRetrieveWrapper<T>  
{  
    private T storedData;  
  
    void IStoreWrapper<T>.SetData(T data)  
    {  
        this.storedData = data;  
    }  
  
    T IRetrieveWrapper<T>.GetData()  
    {  
        return this.storedData;  
    }  
}
```

Functionally, the `Wrapper<T>` class is the same as before, except that you access the `SetData` and `GetData` methods through different interfaces.

```
Wrapper<string> stringWrapper = new Wrapper<string>();  
IStoreWrapper<string> storedStringWrapper = stringWrapper;  
storedStringWrapper.SetData("Hello");  
IRetrieveWrapper<string> retrievedStringWrapper = stringWrapper;  
Console.WriteLine($"Stored value is {retrievedStringWrapper.GetData()}");
```

So, is the following code legal?

```
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;
```

The quick answer is no, and it fails to compile with the same error as before. But if you think about it, although the C# compiler has deemed that this statement is not type safe, the reasons for assuming this are no longer valid. The `IRetrieveWrapper<T>` interface only allows you to read the data held in the `Wrapper<T>` object by using the `GetData` method, and it doesn't provide any way to change the data. In situations such as this where the type parameter occurs only as the return value of the methods in a generic interface, you can inform the compiler that some implicit conversions are legal and that it doesn't have to enforce strict type safety. You do this by specifying the `out` keyword when you declare the type parameter, like this:

```
interface IRetrieveWrapper<out T>
{
    T GetData();
}
```

This feature is called *covariance*. You can assign an `IRetrieveWrapper<A>` object to an `IRetrieveWrapper<B>` reference as long as there's a valid conversion from type A to type B, or type A derives from type B. The following code now compiles and runs as expected:

```
// string derives from object, so this is now legal
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;
```

You can specify the `out` qualifier with a type parameter only if the type parameter occurs as the return type of methods. If you use the type parameter to specify the type of any method parameters, the `out` qualifier is illegal, and your code will not compile. Also, covariance works only with reference types. This is because value types cannot form inheritance hierarchies. So, the following code will not compile because `int` is a value type:

```
Wrapper<int> intWrapper = new Wrapper<int>();
IStoreWrapper<int> storedIntWrapper = intWrapper; // this is legal
...
// the following statement is not legal - ints are not objects
IRetrieveWrapper<object> retrievedObjectWrapper = intWrapper;
```

Several of the interfaces defined by the .NET libraries exhibit covariance, including the `IEnumerable<T>` interface, which is detailed in Chapter 19, "Enumerating collections."



Note Only interface and delegate types (covered in Chapter 18) can be declared as covariant. You do not specify the `out` modifier with generic classes.

Contravariant interfaces

Contravariance follows a similar principle to covariance except that it works in the opposite direction. That is, it enables you to use a generic interface to reference an object of type B through a reference to type A as long as type B derives from type A. This sounds complicated, so it's worth looking at an example from the .NET class library.

The `System.Collections.Generic` namespace in .NET provides an interface called `IComparer`, which looks like this:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

A class that implements this interface must define the `Compare` method, which is used to compare two objects of the type specified by the `T` type parameter. The `Compare` method is expected to return an integer value: 0 if parameters `x` and `y` have the same value, negative if `x` is less than `y`, and positive if `x` is greater than `y`.

The following code shows an example that sorts objects according to their hash code. (The `GetHashCode` method is implemented by the `Object` class. It simply returns an integer value that identifies the object. All reference types inherit this method and can override it with their own implementations.)

```
class ObjectComparer : IComparer<Object>
{
    int IComparer<Object>.Compare(Object x, Object y)
    {
        int xHash = x.GetHashCode();
        int yHash = y.GetHashCode();
        if (xHash == yHash) return 0;
        if (xHash < yHash)
            return -1;
        return 1;
    }
}
```

You can create an `ObjectComparer` object and call the `Compare` method through the `IComparer<Object>` interface to compare two objects, like this:

```
Object x = ...;
Object y = ...;
ObjectComparer objectComparer = new ObjectComparer();
IComparer<Object> objectComparator = objectComparer;
int result = objectComparator.Compare(x, y);
```

That's the boring part. What's more interesting is that you can reference this same object through a version of the `IComparer` interface that compares strings, like this:

```
IComparer<String> stringComparator = objectComparer;
```

At first glance, this statement seems to break every rule of type safety that you can imagine. However, if you think about what the `IComparer<T>` interface does, this approach makes sense. The purpose of the `Compare` method is to return a value based on a comparison between the parameters passed in. If you can compare `Objects`, you certainly should be able to compare `Strings`, which are just specialized types of `Objects`. After all, a `String` should be able to do anything that an `Object` can do—that's the purpose of inheritance.

This still sounds a little presumptive, however. How does the C# compiler know that you're not going to perform any type-specific operations in the code for the `Compare` method that might fail if you invoke the method through an interface based on a different type? If you revisit the definition of the `IComparer` interface, you can see the `in` qualifier before the type parameter:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

The `in` keyword tells the C# compiler that you can either pass the type `T` as the parameter type to methods or pass any type that derives from `T`. You cannot use `T` as the return type from any methods. Essentially, this makes it possible for you to reference an object either through a generic interface based on the object type or through a generic interface based on a type that derives from the object type. Basically, if type `A` exposes some operations, properties, or fields, and if type `B` derives from type `A`, then type `B` must also expose the same operations (which might behave differently if they have been overridden), properties, and fields. Consequently, it should be safe to substitute an object of type `B` for an object of type `A`.

Covariance and contravariance might seem like fringe topics in the world of generics, but they are useful. For example, the `List<T>` generic collection class (in the `System.Collections.Generic` namespace) uses `IComparer<T>` objects to implement the `Sort` and `BinarySearch` methods. A `List<Object>` object can contain a collection of objects of any type, so the `Sort` and `BinarySearch` methods need to be able to sort objects of any type. Without using contravariance, the `Sort` and `BinarySearch` methods would need to include logic that determines the real types of the items being sorted or searched and then implement a type-specific sort or search mechanism. However, unless you're a mathematician, it can be quite difficult to recall what covariance and contravariance actually do. The way I remember, based on the examples in this section, is as follows:

- **Covariance example** If the methods in a generic interface can return strings, they can also return objects. (All strings are objects.)
- **Contravariance example** If the methods in a generic interface can take object parameters, they can take string parameters. (If you can perform an operation by using an object, you can perform the same operation by using a string because all strings are objects.)



Note As with covariance, only interface and delegate types can be declared as contravariant. You don't specify the `in` modifier with generic classes.

Summary

In this chapter, you learned how to use generics to create type-safe classes. You saw how to instantiate a generic type by specifying a type parameter. You also saw how to implement a generic interface and define a generic method. Finally, you learned how to define covariant and contravariant generic interfaces that can operate with a hierarchy of types.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 18.
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

| To | Do this |
|--|---|
| Instantiate an object by using a generic type | Specify the appropriate generic type parameter. For example:
<code>Queue<int> myQueue = new Queue<int>();</code> |
| Create a new generic type | Define the class by using a type parameter. For example:
<pre>public class Tree<TItem> { ... }</pre> |
| Restrict the type that can be substituted for the generic type parameter | Specify a constraint by using a where clause when defining the class. For example:
<pre>public class Tree<TItem> where TItem : IComparable<TItem> { ... }</pre> |
| Define a generic method | Define the method by using type parameters. For example:
<pre>static void InsertIntoTree<TItem> (Tree<TItem> tree, params TItem[] data) { ... }</pre> |
| Invoke a generic method | Provide types for each of the type parameters. For example:
<code>InsertIntoTree<char>(charTree, 'Z', 'X');</code> |
| Define a covariant interface | Specify the out qualifier for covariant type parameters. Reference the covariant type parameters only as the return types from methods and not as the types for method parameters:
<pre>interface IRetrieveWrapper<out T> { T GetData(); }</pre> |
| Define a contravariant interface | Specify the in qualifier for contravariant type parameters. Reference the contravariant type parameters only as the types of method parameters and not as return types:
<pre>public interface IComparer<in T> { int Compare(T x, T y); }</pre> |

Using collections

After completing this chapter, you will be able to:

- Explain the functionality provided in the different collection classes available with .NET and understand how to use them.
- Use collection initializers to populate collections.
- Search through collections to find matching data.
- Understand the differences between collections and arrays.

Chapter 10, “Using arrays,” introduced arrays for holding sets of data. Arrays are very useful in this respect, but they have their drawbacks. Arrays provide only limited functionality; for example, it’s not easy to increase or reduce the size of an array, and neither is it a simple matter to sort the data held in an array. Also, arrays only really provide a single means of accessing data: by using an integer index. If your application needs to use some other mechanism to store and retrieve data, such as the first-in, first-out queue mechanism described in Chapter 17, “Introducing generics,” arrays might not be the most suitable data structure. This is where collections can prove useful.

What are collection classes?

Microsoft .NET provides several classes that collect elements together such that an application can access the elements in specialized ways. These are the collection classes mentioned in Chapter 17, and they live in the `System.Collections.Generic` namespace.

As the namespace implies, these collections are generic types; they expect you to provide a type parameter indicating the kind of data that your application will store in them. Each collection class is optimized for a particular form of data storage and access, and each provides specialized methods that support this functionality. For example, the `Stack<T>` class implements a last-in, first-out model, where you add an item to the top of the stack by using the `Push` method, and you take an item from the top of the stack by using the `Pop` method. The `Pop` method always retrieves the most recently pushed item and removes it from the stack. In contrast, the `Queue<T>` type provides the `Enqueue` and `Dequeue` methods described in Chapter 17. The `Enqueue` method adds an item to the queue, while the `Dequeue` method retrieves items from the queue in the same order, implementing a first-in, first-out model. A variety of other collection classes are also available; the following table summarizes the most commonly used ones.

| Collection | Description |
|------------------------------------|--|
| List<T> | A list of objects that can be accessed by index, as with an array, but with additional methods with which to search the list and sort the contents of the list. |
| Queue<T> | A first-in, first-out data structure, with methods to add an item to one end of the queue, remove an item from the other end, and examine an item without removing it. |
| PriorityQueue<TElement, TPriority> | A queue that sorts data by priority. Elements of a higher priority are dequeued before those of a lower priority. Elements with the same priority are dequeued in the order in which they were enqueued. |
| Stack<T> | A first-in, last-out data structure with methods to push an item onto the top of the stack, pop an item from the top of the stack, and examine the item at the top of the stack without removing it. |
| LinkedList<T> | A double-ended ordered list, optimized to support insertion and removal at either end. This collection can act like a queue or a stack, but it also supports random access as a list does. |
| HashSet<T> | An unordered set of values that is optimized for fast retrieval of data. It provides set-oriented methods for determining whether the items it holds are a subset of those in another HashSet<T> object as well as computing the intersection and union of HashSet<T> objects. |
| Dictionary<TKey, TValue> | A collection of values that can be identified and retrieved by using keys rather than indexes. |
| SortedList<TKey, TValue> | A sorted list of key/value pairs. The keys must implement the IComparable<T> interface. |

The following sections provide a brief overview of these collection classes. Refer to the .NET class library documentation for more details on each class.

.NET class library collection types

The .NET class library provides another set of collection types in the `System.Collections` namespace. These are non-generic collections, and they were designed before C# supported generic types. (Generics were added to the version of C# developed for .NET version 2.0.) With one exception, these types all store object references, and you are required to perform the appropriate casts when you store and retrieve items. These classes are included for backward compatibility with existing applications, and it is not recommended that you use them when building new solutions. In fact, these classes are not available if you're building Universal Windows Platform (UWP) apps.

The one class that does not store object references is the `BitArray` class. This class implements a compact array of Boolean values by using an `int`; each bit indicates true (1) or false (0). If this sounds familiar, it should; this is very similar to the `IntBits` struct that you saw in Chapter 16, "Handling binary data and using indexers." The `BitArray` class is available to UWP apps.

One other important set of collections is available, which are defined in the `System.Collections.Concurrent` namespace. These are thread-safe collection classes that you can use when you're building multithreaded applications. Chapter 24, "Improving response time by performing asynchronous operations," provides more information on these classes.

The List<T> collection class

The generic `List<T>` class is the simplest of the collection classes. You can use it much like you use an array. You can reference an existing element in a `List<T>` collection by using ordinary array notation, with square brackets and the index of the element, although you can't use array notation to add new elements. However, in general, the `List<T>` class provides more flexibility than arrays do and is designed to overcome the following restrictions exhibited by arrays:

- If you want to resize an array, you must create a new array, copy the elements (leaving out some if the new array is smaller), and then update any references to the original array so that they refer to the new array.
- If you want to remove an element from an array, you must move all the trailing elements up by one place. Even this doesn't quite work because you end up with two copies of the last element.
- If you want to insert an element into an array, you must move elements down by one place to make a free slot. However, you lose the last element of the array!

The following features of the `List<T>` collection class address these limitations:

- You don't need to specify the capacity of a `List<T>` collection when you create it; it can grow and shrink as you add elements. There's an overhead associated with this dynamic behavior, and if necessary, you can specify an initial size. However, if you exceed this size, the `List<T>` collection simply grows as necessary.
- You can remove a specified element from a `List<T>` collection by using the `Remove` method. The `List<T>` collection automatically reorders its elements and closes the gap. You can also remove an item at a specified position in a `List<T>` collection by using the `RemoveAt` method.
- You can add an element to the end of a `List<T>` collection by using its `Add` method. You supply the element to be added. The `List<T>` collection resizes itself automatically.
- You can insert an element into the middle of a `List<T>` collection by using the `Insert` method. Again, the `List<T>` collection resizes itself.
- You can easily sort the data in a `List<T>` object by calling the `Sort` method.



Note As with arrays, if you use `foreach` to iterate through a `List<T>` collection, you cannot use the iteration variable to modify the contents of the collection. Additionally, you cannot call the `Remove`, `Add`, or `Insert` method in a `foreach` loop that iterates through a `List<T>` collection; any attempt to do so results in an `InvalidOperationException` exception.

Here's an example that shows how you can create, manipulate, and iterate through the contents of a `List<int>` collection:

```
using System;
using System.Collections.Generic;
...
List<int> numbers = new List<int>();
```

```
// Fill the List<int> by using the Add method
foreach (int number in new int[12]{10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1})
{
    numbers.Add(number);
}

// Insert an element in the penultimate position in the list, and move the last item up
// The first parameter is the position; the second parameter is the value being inserted
numbers.Insert(numbers.Count-1, 99);

// Remove the first element whose value is 7 (the 4th element, index 3)
numbers.Remove(7);

// Remove the element that's now the 7th element
numbers.RemoveAt(6);

// Iterate the remaining 11 elements using a for statement
Console.WriteLine("Iterating using a for statement:");
for (int i = 0; i < numbers.Count; i++)
{
    int number = numbers[i]; // Note the use of array syntax
    Console.WriteLine(number);
}

// Iterate the same 11 elements using a foreach statement
Console.WriteLine("\n Iterating using a foreach statement:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

Here's the output of this code:

```
Iterating using a for statement:
10
9
8
7
6
5
4
3
2
99
1
Iterating using a foreach statement:
10
9
8
7
6
5
4
3
2
99
1
```



Note The way you determine the number of elements for a `List<T>` collection is different from querying the number of items in an array. When using a `List<T>` collection, you examine the `Count` property; when using an array, you examine the `Length` property.

The `LinkedList<T>` collection class

The `LinkedList<T>` collection class implements a doubly linked list. Each item in the list holds the value for that item together with a reference to the next item in the list (the `Next` property) and the previous item (the `Previous` property). The item at the start of the list has the `Previous` property set to `null`, and the item at the end of the list has the `Next` property set to `null`.

Unlike the `List<T>` class, `LinkedList<T>` doesn't support array notation for inserting or examining elements. Instead, you can use the `AddFirst` method to insert an element at the start of the list, moving the previous first item up and setting its `Previous` property to refer to the new item. Similarly, you can use the `AddLast` method to insert an element at the end of the list, setting the `Next` property of the previously last item to refer to the new item. You can also use the `AddBefore` and `AddAfter` methods to insert an element before or after a specified item in the list. (You must retrieve the item first.)

You can find the first item in a `LinkedList<T>` collection by querying the `First` property, and querying the `Last` property returns a reference to the final item in the list. To iterate through a linked list, you can start at one end and step through the `Next` or `Previous` references until you find an item with a `null` value for this property. Alternatively, you can use a `foreach` statement, which iterates forward through a `LinkedList<T>` object and stops automatically at the end. You delete an item from a `LinkedList<T>` collection by using the `Remove`, `RemoveFirst`, and `RemoveLast` methods.

The following example shows a `LinkedList<T>` collection in action. Notice how the code that iterates through the list by using a `for` statement steps through the `Next` (or `Previous`) references, stopping only when it reaches a `null` reference, which is the end of the list:

```
using System;
using System.Collections.Generic;
...
LinkedList<int> numbers = new LinkedList<int>();

// Fill the List<int> by using the AddFirst method
foreach (int number in new int[] { 10, 8, 6, 4, 2 })
{
    numbers.AddFirst(number);
}

// Iterate using a for statement
Console.WriteLine("Iterating using a for statement:");
for (LinkedListNode<int> node = numbers.First; node is not null; node = node.Next)
{
    int number = node.Value;
    Console.WriteLine(number);
}
```

```
// Iterate using a foreach statement
Console.WriteLine("\nIterating using a foreach statement:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

// Iterate backward
Console.WriteLine("\n Iterating list in reverse order:");
for (LinkedListNode<int> node = numbers.Last; node is not null; node = node.Previous)
{
    int number = node.Value;
    Console.WriteLine(number);
}
```

Here's the output generated by this code:

```
Iterating using a for statement:
2
4
6
8
10
Iterating using a foreach statement:
2
4
6
8
10
Iterating list in reverse order:
10
8
6
4
2
```

The Queue<T> collection class

The Queue<T> class implements a first-in, first-out mechanism. An element is inserted into the queue at the back (the Enqueue operation) and is removed from the queue at the front (the Dequeue operation).

The following code is an example showing a Queue<int> collection and its common operations:

```
using System;
using System.Collections.Generic;
...
Queue<int> numbers = new Queue<int>();

// fill the queue
Console.WriteLine("Populating the queue:");
foreach (int number in new int[4]{9, 3, 7, 2})
{
    numbers.Enqueue(number);
    Console.WriteLine($"{number} has joined the queue");
}
```

```
// iterate through the queue
Console.WriteLine("\n The queue contains the following items:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

// empty the queue
Console.WriteLine("\n Draining the queue:");
while (numbers.Count > 0)
{
    int number = numbers.Dequeue();
    Console.WriteLine($"{number} has left the queue");
}
```

Here's the output from this code:

```
Populating the queue:
9 has joined the queue
3 has joined the queue
7 has joined the queue
2 has joined the queue
The queue contains the following items:
9
3
7
2
Draining the queue:
9 has left the queue
3 has left the queue
7 has left the queue
2 has left the queue
```

The PriorityQueue<TElement, TPriority> collection class

The `PriorityQueue<TElement, TPriority>` class extends the idea of the `Queue<T>` class to associate a priority with each queued element. The class takes two type parameters: the first is the type of the elements that are queued, and the second indicates the type for the priority (commonly an `int`, but it can be any comparable type). Elements are enqueued at the back of the queue but can be dequeued earlier than longer-lived elements if they have a higher priority than those elements. All elements with the same priority are enqueued and dequeued in the same order, as for a regular queue.

The following code shows an example, illustrating how elements with different priorities are enqueued and dequeued. Elements with a lower-priority value have a higher priority (elements with priority 1 are dequeued before those with priority 2):

```
using System;
using System.Collections.Generic;
...
// Create the queue
// The data items are strings. The second type parameter (an int) indicates the priority
PriorityQueue<string, int> messages = new PriorityQueue<string, int>();

// Add messages with different priorities to the queue
messages.Enqueue("Twas", 1);
```



```

messages.Enqueue("Brillig", 1);
messages.Enqueue("and", 2);
messages.Enqueue("the", 3);
messages.Enqueue("Slithy", 2);
messages.Enqueue("Toves", 3);

// Retrieve the messages in priority and queue order
while (messages.TryDequeue(out string item, out int priority))
{
    Console.WriteLine($"Dequeued Item : {item} Priority Was : {priority}");
}

```

In the output, messages with priority 1 appear first, followed by those with priority 2, and finally those with priority 3.

```

Popped Item : Twas. Priority Was : 1
Popped Item : Brillig. Priority Was : 1
Popped Item : Slithy. Priority Was : 2
Popped Item : and. Priority Was : 2
Popped Item : the. Priority Was : 3
Popped Item : Toves. Priority Was : 3

```

The Stack<T> collection class

The Stack<T> class implements a last-in, first-out mechanism. An element joins the stack at the top (the push operation) and leaves the stack at the top (the pop operation). To visualize this, think of a stack of dishes. New dishes are added to the top, and dishes are removed from the top, making the last dish to be placed on the stack the first one to be removed. (The dish at the bottom is rarely used and will inevitably require washing before you can put any food on it because it will be covered in grime!) Here's an example; notice the order in which the items are listed by the foreach loop:

```

using System;
using System.Collections.Generic;
...
Stack<int> numbers = new Stack<int>();

// fill the stack
Console.WriteLine("Pushing items onto the stack:");
foreach (int number in new int[4]{9, 3, 7, 2})
{
    numbers.Push(number);
    Console.WriteLine($"{number} has been pushed on the stack");
}

// iterate through the stack
Console.WriteLine("\n The stack now contains:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

// empty the stack
Console.WriteLine("\n Popping items from the stack:");
while (numbers.Count > 0)
{

```

```

        int number = numbers.Pop();
        Console.WriteLine($"{number} has been popped off the stack");
    }

```

Here's the output from this program:

```

Pushing items onto the stack:
9 has been pushed on the stack
3 has been pushed on the stack
7 has been pushed on the stack
2 has been pushed on the stack
The stack now contains:
2
7
3
9
Popping items from the stack:
2 has been popped off the stack
7 has been popped off the stack
3 has been popped off the stack
9 has been popped off the stack

```

The Dictionary<TKey, TValue> collection class

The array and `List<T>` types provide a way to map an integer index to an element. You specify an integer index within square brackets (for example, `[4]`), and you get back the element at index (4, which is actually the fifth element). However, sometimes you might want to implement a mapping in which the type from which you map is not an `int` but some other type, such as `string`, `double`, or `Time`. In other languages, this is often called an *associative array*.

The `Dictionary<TKey, TValue>` class implements this functionality by internally maintaining two arrays—one for the keys you're mapping from and one for the values you're mapping to. When you insert a key/value pair into a `Dictionary<TKey, TValue>` collection, it automatically tracks which key belongs to which value and makes it possible for you to retrieve the value that is associated with a specified key quickly and easily.

The design of the `Dictionary<TKey, TValue>` class has some important consequences:

- A `Dictionary<TKey, TValue>` collection cannot contain duplicate keys. If you call the `Add` method to add a key that's already present in the keys array, you'll get an exception. You can, however, use the square brackets notation to add a key/value pair (as shown in the following example) without danger of an exception, even if the key has already been added; any existing value with the same key will be overwritten by the new value. You can test whether a `Dictionary<TKey, TValue>` collection already contains a particular key by using the `ContainsKey` method.
- Internally, a `Dictionary<TKey, TValue>` collection is a sparse data structure that operates most efficiently when it has plenty of memory to work with. The size of a `Dictionary<TKey, TValue>` collection in memory can grow quite quickly as you insert more elements.

- When you use a `foreach` statement to iterate through a `Dictionary<TKey, TValue>` collection, you get back a `KeyValuePair<TKey, TValue>` item. This is a structure that contains a copy of the key and value elements of an item in the `Dictionary<TKey, TValue>` collection, and you can access each element through the `Key` property and the `Value` property. These elements are read-only; you can't use them to modify the data in the `Dictionary<TKey, TValue>` collection.

Here's an example that associates the ages of members of my family with their names and then prints the information:

```
using System;
using System.Collections.Generic;
...
Dictionary<string, int> ages = new Dictionary<string, int>();

// fill the Dictionary
ages.Add("John", 57); // using the Add method
ages.Add("Diana", 57);
ages["James"] = 30; // using array notation
ages["Francesca"] = 27;

// iterate using a foreach statement
// the iterator generates a KeyValuePair item
Console.WriteLine("The Dictionary contains:");
foreach (KeyValuePair<string, int> element in ages)
{
    string name = element.Key;
    int age = element.Value;
    Console.WriteLine($"Name: {name}, Age: {age}");
}
```

Here's the output from this program:

```
The Dictionary contains:
Name: John, Age: 57
Name: Diana, Age: 57
Name: James, Age: 30
Name: Francesca, Age: 27
```



Note The `System.Collections.Generic` namespace also includes the `SortedDictionary<TKey, TValue>` collection type. This class maintains the collection in order, sorted by the keys.

The `SortedList<TKey, TValue>` collection class

The `SortedList<TKey, TValue>` class is very similar to the `Dictionary<TKey, TValue>` class in that you can use it to associate keys with values. The primary difference is that the keys array is always sorted. (It's called a `SortedList`, after all.) In most cases, it takes longer to insert data into a `SortedList<TKey, TValue>` object than a `SortedDictionary<TKey, TValue>` object, but data retrieval is often quicker (or at least as quick), and the `SortedList<TKey, TValue>` class uses less memory.

When you insert a key/value pair into a `SortedList<TKey, TValue>` collection, the key is inserted into the keys array at the correct index to keep the keys array sorted. The value is then inserted into the values array at the same index. The `SortedList<TKey, TValue>` class automatically ensures that keys and values remain synchronized, even when you add and remove elements. This means you can insert key/value pairs into a `SortedList<TKey, TValue>` in any sequence; they are always sorted based on the value of the keys.

Like the `Dictionary<TKey, TValue>` class, a `SortedList<TKey, TValue>` collection cannot contain duplicate keys. When you use a `foreach` statement to iterate through a `SortedList<TKey, TValue>`, you receive a `KeyValuePair<TKey, TValue>` item. However, the `KeyValuePair<TKey, TValue>` items will be returned sorted by the `Key` property.

Here's the same example that associates the ages of members of my family with their names and then prints the information, but this version has been adjusted to use a `SortedList<TKey, TValue>` object rather than a `Dictionary<TKey, TValue>` collection:

```
using System;
using System.Collections.Generic;
...
SortedList<string, int> ages = new SortedList<string, int>();

// fill the SortedList
ages.Add("John", 57);    // using the Add method
ages.Add("Diana", 57);
ages["James"] = 30;     // using array notation
ages["Francesca"] = 27;

// iterate using a foreach statement
// the iterator generates a KeyValuePair item
Console.WriteLine("The SortedList contains:");
foreach (KeyValuePair<string, int> element in ages)
{
    string name = element.Key;
    int age = element.Value;
    Console.WriteLine($"Name: {name}, Age: {age}");
}
```

The output from this program is sorted alphabetically by the names of my family members:

```
The SortedList contains:
Name: Diana, Age: 57
Name: Francesca, Age: 27
Name: James, Age: 30
Name: John, Age: 57
```

The `HashSet<T>` collection class

The `HashSet<T>` class is optimized for performing set operations, such as determining set membership and generating the union and intersection of sets.

You insert items into a `HashSet<T>` collection by using the `Add` method, and you delete items by using the `Remove` method. However, the real power of the `HashSet<T>` class is provided by the `IntersectWith`, `UnionWith`, and `ExceptWith` methods. These methods modify a `HashSet<T>`

collection to generate a new set that either intersects with, has a union with, or does not contain the items in a specified `HashSet<T>` collection. These operations are destructive in as much as they overwrite the contents of the original `HashSet<T>` object with the new set of data.

You can also determine whether the data in one `HashSet<T>` collection is a superset or subset of another by using the `IsSubsetOf`, `IsSupersetOf`, `IsProperSubsetOf`, and `IsProperSupersetOf` methods. These methods return a Boolean value and are nondestructive.

Internally, a `HashSet<T>` collection is held as a hash table, enabling the fast lookup of items. However, a large `HashSet<T>` collection can require a significant amount of memory to operate quickly.

The following example shows how to populate a `HashSet<T>` collection and illustrates the use of the `IntersectWith` method to find data that overlaps two sets:

```
using System;
using System.Collections.Generic;
...
HashSet<string> employees = new HashSet<string>(new string[] { "Fred", "Bert", "Harry", "John" });
HashSet<string> customers = new HashSet<string>(new string[] { "John", "Sid", "Harry", "Diana" });

employees.Add("James");
customers.Add("Francesca");

Console.WriteLine("Employees:");
foreach (string name in employees)
{
    Console.WriteLine(name);
}

Console.WriteLine("");
Console.WriteLine("Customers:");
foreach (string name in customers)
{
    Console.WriteLine(name);
}

Console.WriteLine("\nCustomers who are also employees:");
customers.IntersectWith(employees);
foreach (string name in customers)
{
    Console.WriteLine(name);
}
```

This code generates the following output:

```
Employees:
Fred
Bert
Harry
John
James

Customers:
John
Sid
Harry
```

Diana
Francesca
Customers who are also employees:
John
Harry



Note The `System.Collections.Generic` namespace also provides the `SortedSet<T>` collection type, which operates similarly to the `HashSet<T>` class. The primary difference, as the name implies, is that the data is maintained in a sorted order. The `SortedSet<T>` and `HashSet<T>` classes are interoperable; you can take the union of a `SortedSet<T>` collection with a `HashSet<T>` collection, for example.

Using collection initializers

The examples in the preceding section have shown you how to add individual elements to a collection by using the method most appropriate to that collection (`Add` for a `List<T>` collection, `Enqueue` for a `Queue<T>` collection, `Push` for a `Stack<T>` collection, and so on). You can also initialize some collection types when you declare them by using a syntax similar to that supported by arrays. For example, the following statement creates and initializes the `numbers List<int>` object shown earlier, demonstrating an alternative to repeatedly calling the `Add` method:

```
List<int> numbers = new List<int>(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};
```

Internally, the C# compiler converts this initialization to a series of calls to the `Add` method. Consequently, you can use this syntax only for collections that actually support the `Add` method. (The `Stack<T>` and `Queue<T>` classes do not.)

For more complex collections that take key/value pairs, such as the `Dictionary<TKey, TValue>` class, you can use indexer notation to specify a value for each key, like this:

```
Dictionary<string, int> ages = new Dictionary<string, int>()
{
    ["John"] = 53,
    ["Diana"] = 53,
    ["James"] = 26,
    ["Francesca"] = 23
};
```

If you prefer, you can also specify each key/value pair as an anonymous type in the initializer list, like this:

```
Dictionary<string, int> ages = new Dictionary<string, int>()
{
    {"John", 53},
    {"Diana", 53},
    {"James", 26},
    {"Francesca", 23}
};
```

In this case, the first item in each pair is the key, and the second is the value. To make your code as readable as possible, I recommend that you use the indexer notation wherever you can when you initialize a dictionary type.

Find methods, predicates, and lambda expressions

Using dictionary-oriented collections (`Dictionary<TKey, TValue>`, `SortedDictionary<TKey, TValue>`, and `SortedList<TKey, TValue>`), you can quickly find a value by specifying the key to search for, and you can use array notation to access the value, as you have seen in earlier examples. Other collections that support nonkeyed random access, such as the `List<T>` and `LinkedList<T>` classes, do not support array notation but instead provide the `Find` method to locate an item.

For these classes, the argument to the `Find` method is a predicate that specifies the search criteria to use. The form of a predicate is a method that examines each item in the collection and returns a Boolean value indicating whether the item matches. In the case of the `Find` method, as soon as the first match is found, the corresponding item is returned. Note that the `List<T>` and `LinkedList<T>` classes also support other methods, such as `FindLast`, which returns the last matching object, and the `List<T>` class additionally provides the `FindAll` method, which returns a `List<T>` collection of all matching objects.

The easiest way to specify the predicate is to use a *lambda expression*. A lambda expression is an expression that returns a method. This sounds rather odd because most expressions that you've encountered so far in C# actually return a value. If you're familiar with functional programming languages such as Haskell, you're probably comfortable with this concept. If you're not, fear not. Lambda expressions are not particularly complicated, and after you become accustomed to a new bit of syntax, you'll see that they are very useful.



Note If you're interested in finding out more about functional programming with Haskell, visit the Haskell programming language website at <http://www.haskell.org/haskellwiki/Haskell>.

Chapter 3, "Writing methods and applying scope," explained that a typical method consists of four elements: a return type, a method name, a list of parameters, and a method body. A lambda expression contains two of these elements: a list of parameters and a method body. Lambda expressions do not define a method name, and the return type (if any) is inferred from the context in which the lambda expression is used. In the case of the `Find` method, the predicate processes each item in the collection in turn; the body of the predicate must examine the item and return `true` or `false` depending on whether it matches the search criteria. The example that follows shows the `Find` method (highlighted in bold) on a `List<Person>` collection, where `Person` is a struct. The `Find` method returns the first item in the list that has the `ID` property set to 3:

```
struct Person
{
    public int ID { get; set; }
    public string Name { get; set; }
```

```

    public int Age { get; set; }
}
...
// Create and populate the personnel list
List<Person> personnel = new List<Person>()
{
    new Person() { ID = 1, Name = "John", Age = 53 },
    new Person() { ID = 2, Name = "Sid", Age = 28 },
    new Person() { ID = 3, Name = "Fred", Age = 34 },
    new Person() { ID = 4, Name = "Paul", Age = 22 },
};

// Find the member of the list that has an ID of 3
Person match = personnel.Find((Person p) => { return p.ID == 3; });
Console.WriteLine($"ID: {match.ID}\nName: {match.Name}\nAge: {match.Age}");

```

Here's the output generated by this code:

```

ID: 3
Name: Fred
Age: 34

```

In the call to the `Find` method, the `(Person p) => { return p.ID == 3; }` argument is a lambda expression that actually does the work. It has the following syntactic items:

- **A list of parameters enclosed in parentheses** As with a regular method, if the method you are defining (as in the preceding example) takes no parameters, you must still provide the parentheses. In the case of the `Find` method, the predicate is provided with each item from the collection in turn, and this item is passed as the parameter to the lambda expression.
- **The `=>` operator** This indicates to the C# compiler that this is a lambda expression.
- **The body of the method** The example shown here is very simple, containing a single statement that returns a Boolean value indicating whether the item specified in the parameter matches the search criteria. However, a lambda expression can contain multiple statements, and you can format it in whatever way you feel is most readable. Just remember to add a semicolon after each statement, as you would in an ordinary method.



Important You also saw in Chapter 3 how the `=>` operator is used to define expression-bodied methods. Rather confusingly, this is a somewhat overloaded use of the `=>` operator. Although there are some notional similarities, expression-bodied methods and lambda expressions are semantically (and functionally) quite different beasts; you should not confuse them.

Strictly speaking, the body of a lambda expression can be a method body containing multiple statements, or it can be a single expression. If the body of a lambda expression contains only a single expression, you can omit the braces and the semicolon (but you still need a semicolon to complete the entire statement). Additionally, if the expression takes a single parameter, you can omit the parentheses that surround the parameter. Finally, in many cases, you can actually omit the type of the parameters

because the compiler can infer this information from the context from which the lambda expression is invoked. A simplified form of the `Find` statement shown previously looks like this (which is much easier to read and understand):

```
Person match = personnel.Find(p => p.ID == 3);
```



Note The `Stack<T>`, `Queue<T>`, and `HashSet<T>` collection classes do not support searching, although you can test for membership of an item in a hash set by using the `Contains` method.

The forms of lambda expressions

Lambda expressions are very powerful constructs, and you'll encounter them with increasing frequency as you delve deeper into C# programming. The expressions themselves can take some subtly different forms. Lambda expressions were originally part of a mathematical notation called *lambda calculus*, which provides a notation for describing functions. (You can think of a function as a method that returns a value.) Although C# has extended the syntax and semantics of lambda calculus in its implementation of lambda expressions, many of the original principles still apply. Here are some examples showing the different forms of lambda expressions available in C#:

```
x => x * x           // A simple expression that returns the square of its parameter
                    // The type of parameter x is inferred from the context.

x => { return x * x ; } // Semantically the same as the preceding
                    // expression, but using a C# statement block as
                    // a body rather than a simple expression.

(int x) => x / 2      // A simple expression that returns the value of the
                    // parameter divided by 2.
                    // The type of parameter x is stated explicitly.

() => folder.StopFolding(0) // Calling a method
                    // The expression takes no parameters.
                    // The expression might or might not
                    // return a value.

(x, y) => { x++; return x / y; } // Multiple parameters; the compiler
                    // infers the parameter types.
                    // The parameter x is passed by value, so
                    // the effect of the ++ operation is
                    // local to the expression.

(ref int x, int y) => { x++; return x / y; } // Multiple parameters
                    // with explicit types.
                    // Parameter x is passed by
                    // reference, so the effect of
                    // the ++ operation is permanent.
```

To summarize, here are some features of lambda expressions that you should be aware of:

- If a lambda expression takes parameters, you specify them in the parentheses to the left of the `=>` operator. If you omit the types of parameters, the C# compiler will infer their types from the context of the lambda expression. You can pass parameters by reference (by using the `ref` keyword) if you want the lambda expression to be able to change its values other than locally, but this is not recommended.
- Lambda expressions can return values, but the return type must match that of the corresponding delegate.
- The body of a lambda expression can be a simple expression or a block of C# code made up of multiple statements, method calls, variable definitions, and other code items.
- Variables defined in a lambda expression method go out of scope when the method finishes.
- A lambda expression can access and modify all variables outside the lambda expression that are in scope when the lambda expression is defined. Be very careful with this feature!

Lambda expressions and anonymous methods

Lambda expressions were added to the C# language in version 3.0. C# version 2.0 introduced anonymous methods, which can perform a similar task but are not as flexible. Anonymous methods were added primarily so that you can define delegates without having to create a named method; you simply provide the definition of the method body in place of the method name, like this:

```
this.stopMachinery += delegate { folder.StopFolding(0); };
```

You can also pass an anonymous method as a parameter in place of a delegate, like this:

```
control.Add(delegate { folder.StopFolding(0); } );
```

Whenever you introduce an anonymous method, you must prefix it with the `delegate` keyword. Also, any parameters needed are specified in parentheses following the `delegate` keyword, as illustrated in the following example:

```
control.Add(delegate(int param1, string param2)
{ /* code that uses param1 and param2 */ ... });
```

Lambda expressions provide a more succinct and natural syntax than anonymous methods, and they pervade many of the more advanced aspects of C#, as you'll see throughout the subsequent chapters in this book. Generally speaking, you should use lambda expressions rather than anonymous methods in your code.

Comparing arrays and collections

Here's a summary of the important differences between arrays and collections:

- An array instance has a fixed size and cannot grow or shrink. A collection can dynamically resize itself as required.
- An array can have more than one dimension. A collection is linear. However, the items in a collection can be collections themselves, so you can imitate a multidimensional array as a collection of collections.
- You store and retrieve an item in an array by using an index. Not all collections support this notion. For example, to store an item in a `List<T>` collection, you use the `Add` or `Insert` method, and to retrieve an item, you use the `Find` method.
- Many collection classes provide a `ToArray` method that creates and populates an array containing the items in the collection. The items are copied to the array and are not removed from the collection. Additionally, these collections provide constructors that can populate a collection directly from an array.

In the next exercise, you'll convert the card game developed in Chapter 10 to use collections rather than arrays.

To use collections to implement a card game

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **Cards** solution, which is located in the `\Microsoft Press\VCSBS\Chapter 18\Cards` folder in your **Documents** folder.

This project contains an updated version of the project from Chapter 10 that deals hands of cards by using arrays. The `PlayingCard` class is modified to expose the value and suit of a card as read-only properties.

3. Display the `Pack.cs` file in the Code and Text Editor window. Add the following `using` directive to the top of the file:

```
using System.Collections.Generic;
```

4. In the `Pack` class, change the definition of the `cardPack` two-dimensional array to a `Dictionary<Suit, List<PlayingCard>>` object, as shown here in bold:

```
class Pack
{
    ...
    private Dictionary<Suit, List<PlayingCard>> cardPack;
    ...
}
```

The original application used a two-dimensional array to represent a pack of cards. This code replaces the array with a `Dictionary` collection, where the key specifies the suit, and the value is a list of cards in that suit.

5. Locate the `Pack` constructor. Modify the first statement in this constructor to instantiate the `cardPack` variable as a new `Dictionary` collection rather than as an array, as shown here in bold:

```
public Pack()
{
    this.cardPack = new Dictionary<Suit, List<PlayingCard>>(NumSuits);
    ...
}
```

Although a `Dictionary` collection will resize itself automatically as items are added, if the collection is unlikely to change in size, you can specify an initial size when you instantiate it. This helps to optimize the memory allocation (although the `Dictionary` collection can still grow if this size is exceeded). In this case, the `Dictionary` collection will contain a collection of four lists (one list for each suit), so it is allocated space for four items (`NumSuits` is a constant with the value 4).

6. In the outer for loop, declare a `List<PlayingCard>` collection object called `cardsInSuit` that's big enough to hold the number of cards in each suit (use the `CardsPerSuit` constant), as follows in bold:

```
public Pack()
{
    this.cardPack = new Dictionary<Suit, List<PlayingCard>>(NumSuits);
    for (Suit = Suit.Clubs; suit <= Suit.Spades; suit++)
    {
        List<PlayingCard> cardsInSuit = new List<PlayingCard>(CardsPerSuit);
        for (Value value = Value.Two; value <= Value.Ace; value++)
        {
            ...
        }
    }
}
```

7. Change the code in the inner for loop to add new `PlayingCard` objects to this collection rather than to the array, as shown in bold in the following code:

```
for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)
{
    List<PlayingCard> cardsInSuit = new List<PlayingCard>(CardsPerSuit);
    for (Value value = Value.Two; value <= Value.Ace; value++)
    {
        cardsInSuit.Add(new PlayingCard(suit, value));
    }
}
```

8. After the inner for loop, add the `List` object to the `cardPack` `Dictionary` collection, specifying the value of the `suit` variable as the key to this item:

```
for (Suit suit = Suit.Clubs; suit <= Suit.Spades; suit++)
{
    List<PlayingCard> cardsInSuit = new List<PlayingCard>(CardsPerSuit);
    for (Value value = Value.Two; value <= Value.Ace; value++)
    {
        cardsInSuit.Add(new PlayingCard(suit, value));
    }
    this.cardPack.Add(suit, cardsInSuit);
}
```

9. Find the `DealCardFromPack` method.

This method picks a card at random from the pack, removes the card from the pack, and returns this card. The logic for selecting the card does not require any changes, but the statements at the end of the method that retrieve the card from the array must be updated to use the `Dictionary` collection instead. Additionally, the code that removes the card from the array (it has now been dealt) must be modified; you need to search for the card in the list and then remove it from the list. To locate the card, use the `Find` method and specify a predicate that finds a card with the matching value. The parameter to the predicate should be a `PlayingCard` object (the list contains `PlayingCard` items).

The updated statements occur after the closing brace of the second `while` loop, as shown in bold in the following code:

```
public PlayingCard DealCardFromPack()
{
    Suit suit = (Suit)randomCardSelector.Next(NumSuits);
    while (this.IsSuitEmpty(suit))
    {
        suit = (Suit)randomCardSelector.Next(NumSuits);
    }
    Value value = (Value)randomCardSelector.Next(CardsPerSuit);
    while (this.IsCardAlreadyDealt(suit, value))
    {
        value = (Value)randomCardSelector.Next(CardsPerSuit);
    }
    List<PlayingCard> cardsInSuit = this.cardPack[suit];
    PlayingCard card = cardsInSuit.Find(c => c.CardValue == value);
    cardsInSuit.Remove(card);
    return card;
}
```

10. Locate the `IsCardAlreadyDealt` method.

This method determines whether a card has already been dealt by checking whether the corresponding element in the array has been set to `null`. You need to modify this method to determine whether a card with the specified value is present in the list for the suit in the `cardPack` `Dictionary` collection.

To determine whether an item exists in a `List<T>` collection, you use the `Exists` method. This method is similar to `Find` in that it takes a predicate as its argument. The predicate is

passed each item from the collection in turn, and it should return `true` if the item matches some specified criteria, and `false` otherwise. In this case, the `List<T>` collection holds `PlayingCard` objects, and the criteria for the `Exists` predicate should return `true` if it is passed a `PlayingCard` item with a suit and value that matches the parameters passed to the `IsCardAlreadyDealt` method.

Update the method, as shown in the following example in bold:

```
private bool IsCardAlreadyDealt(Suit suit, Value value)
{
    List<PlayingCard> cardsInSuit = this.cardPack[suit];
    return (!cardsInSuit.Exists(c => c.CardSuit == suit && c.CardValue == value));
}
```

11. Display the `Hand.cs` file in the Code and Text Editor window. Add the following `using` directive to the list at the top of the file:

```
using System.Collections.Generic;
```

12. The `Hand` class currently uses an array called `cards` to hold the playing cards for the hand. Modify the definition of the `cards` variable to be a `List<PlayingCard>` collection and comment out the statement that defines the `playingCardCount` variable, as shown here in bold:

```
class Hand
{
    public const int HandSize = 13;
    private List<PlayingCard> cards = new List<PlayingCard>(HandSize);
    // private int playingCardCount = 0;
    ...
}
```

The `playingCardCount` variable was previously used as an index into the array of cards for a hand. This variable is now unnecessary.

13. Find the `AddCardToHand` method.

This method currently checks whether the hand is full; if it is not, it adds the card provided as the parameter to the `cards` array at the index specified by the `playingCardCount` variable. Update this method to use the `Add` method of the `List<PlayingCard>` collection instead.

This change also removes the need to explicitly keep track of how many cards the collection holds because you can use the `Count` property of the `cards` collection instead. Therefore, remove the `playingCardCount` variable from the class and modify the `if` statement that checks whether the hand is full to reference the `Count` property of the `cards` collection.

The completed method should look like this, with the changes highlighted in bold:

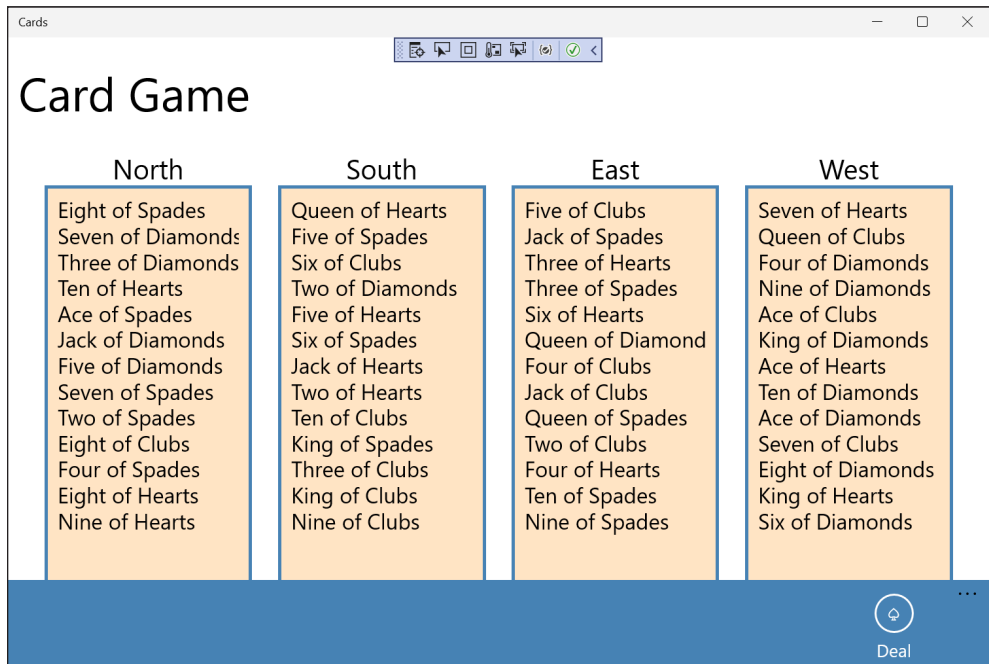
```
public void AddCardToHand(PlayingCard cardDealt)
{
    if (this.cards.Count >= HandSize)
    {
        throw new ArgumentException("Too many cards");
    }
    this.cards.Add(cardDealt);
}
```

14. On the **Debug** menu, select **Start Debugging** to build and run the application.
15. When the Card Game form appears, select **Deal**.



Note The Deal button is located on the command bar. You may need to expand the command bar to reveal the button.

16. Verify that the cards are dealt and that the populated hands appear as before. Then select **Deal** again to generate another random set of hands. The following image shows the application running:



17. Return to Visual Studio 2022 and stop debugging.

Summary

In this chapter, you learned how to use some of the common collection classes to store and access data. In particular, you learned how to use generic collection classes to create type-safe collections. You also learned how to create lambda expressions to search for specific items within collections.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 19, “Enumerating collections.”
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

| To | Do this |
|--|--|
| Create a new collection | Use the constructor for the collection class. For example:
<code>List<PlayingCard> cards = new List<PlayingCard>();</code> |
| Add an item to a collection | Use the Add or Insert methods (as appropriate) for lists, hash sets, and dictionary-oriented collections. Use the Enqueue method for Queue<T> collections. Use the Push method for Stack<T> collections. For example:
<code>HashSet<string> employees = new HashSet<string>(); employees.Add("John");</code>
...
<code>LinkedList<int> data = new LinkedList<int>(); data.AddFirst(101);</code>
...
<code>Stack<int> numbers = new Stack<int>(); numbers.Push(99);</code> |
| Remove an item from a collection | Use the Remove method for lists, hash sets, and dictionary-oriented collections. Use the Dequeue method for Queue<T> collections. Use the Pop method for Stack<T> collections. For example:
<code>HashSet<string> employees = new HashSet<string>(); employees.Remove("John");</code>
...
<code>LinkedList<int> data = new LinkedList<int>(); data.Remove(101);</code>
...
<code>Stack<int> numbers = new Stack<int>();</code>
...
<code>int item = numbers.Pop();</code> |
| Find the number of elements in a collection | Use the Count property. For example:
<code>List<PlayingCard> cards = new List<PlayingCard>();</code>
...
<code>int noOfCards = cards.Count;</code> |
| Locate an item in a collection | For dictionary-oriented collections, use array notation. For lists, use the Find method. For example:
<code>Dictionary<string, int> ages = new Dictionary<string, int>();</code>
<code>ages.Add("John", 47);</code>
<code>int johnsAge = ages["John"];</code>
...
<code>List<Person> personnel = new List<Person>();</code>
<code>Person match = personnel.Find(p => p.ID == 3);</code> |
| Iterate through the elements of a collection | Use a for statement or a foreach statement. For example:
<code>LinkedList<int> numbers = new LinkedList<int>();</code>
...
<code>for (LinkedListNode<int> node = numbers.First; node is not null; node = node.Next)</code>
<code>{</code>
<code>int number = node.Value;</code>
<code>Console.WriteLine(number);</code>
<code>}</code>
...
<code>foreach (int number in numbers)</code>
<code>{</code>
<code>Console.WriteLine(number);</code>
<code>}</code> |

Enumerating collections

After completing this chapter, you will be able to:

- Manually define an enumerator that you can use to iterate over the elements in a collection.
- Implement an enumerator automatically by creating an iterator.
- Provide additional iterators that can step through the elements of a collection in different sequences.

Chapter 10, “Using arrays,” and Chapter 18, “Using collections,” showed how to use arrays and collection classes to hold sequences or sets of data. Chapter 10 also detailed the `foreach` statement, which you can use to step through, or iterate over, the elements in a collection. In those chapters, you used the `foreach` statement as a quick and convenient way of accessing the contents of an array or a collection, but now it’s time to learn a little more about how this statement actually works. This topic becomes important when you define your own collection classes, and this chapter describes how you can make collections enumerable.

Enumerating the elements in a collection

Chapter 10 presents an example of using the `foreach` statement to list the items in a simple array. The code looks like this:

```
int[] pins = { 9, 3, 7, 2 };
foreach (int pin in pins)
{
    Console.WriteLine(pin);
}
```

The `foreach` construct provides an elegant mechanism that greatly simplifies the code you need to write, but it can be exercised only under certain circumstances: you can use `foreach` only to step through an *enumerable* collection. What exactly is an enumerable collection? The quick answer is that it’s a collection that implements the `System.Collections.IEnumerable` interface.



Note Remember that all arrays in C# are instances of the `System.Array` class. The `System.Array` class is a collection class that implements the `IEnumerable` interface.

The `IEnumerable` interface contains a single method called `GetEnumerator`:

```
IEnumerator GetEnumerator();
```

The `GetEnumerator` method should return an enumerator object that implements the `System.Collections.IEnumerator` interface. The enumerator object is used for stepping through (enumerating) the elements of the collection. The `IEnumerator` interface specifies the following property and methods:

```
object Current { get; }  
bool MoveNext();  
void Reset();
```

Think of an enumerator as a pointer indicating elements in a list. Initially, the pointer points *before* the first element. You call the `MoveNext` method to move the pointer down to the next (first) item in the list; the `MoveNext` method returns `true` if there's another item and `false` if there isn't. You use the `Current` property to access the item currently pointed to, and you use the `Reset` method to return the pointer back to *before* the first item in the list. By using the `GetEnumerator` method of a collection to create an enumerator, repeatedly calling the `MoveNext` method, and using the enumerator to retrieve the value of the `Current` property, you can move forward through the elements of a collection one item at a time. This is exactly what the `foreach` statement does. So, if you want to create your own enumerable collection class, you must implement the `IEnumerable` interface in your collection class and provide an implementation of the `IEnumerator` interface to be returned by the `GetEnumerator` method of the collection class.



Important At first glance, it's easy to confuse the `IEnumerable` and `IEnumerator` interfaces because their names are so similar. Be certain not to mix them up.

You might have noticed that the `Current` property of the `IEnumerator` interface exhibits non-type-safe behavior in that it returns an object rather than a specific type. However, you should be pleased to know that the Microsoft .NET class library also provides the generic `IEnumerator<T>` interface, which has a `Current` property that returns a `T` instead. Likewise, there's an `IEnumerable<T>` interface containing a `GetEnumerator` method that returns an `Enumerator<T>` object. Both of these interfaces are defined in the `System.Collections.Generic` namespace. If you're building applications for .NET version 2.0 or later, you should use these generic interfaces rather than the nongeneric versions when you define enumerable collections.

Manually implementing an enumerator

In the following exercise, you'll define a class that implements the generic `IEnumerator<T>` interface and create an enumerator for the binary tree class demonstrated in Chapter 17, "Introducing generics."

Chapter 17 illustrates how easy it is to traverse a binary tree and display its contents. You would, therefore, be inclined to think that defining an enumerator that retrieves each element in a binary tree

in the same order would be a simple matter. Sadly, you would be mistaken. The main problem is that when defining an enumerator, you must remember where you are in the structure so that subsequent calls to the `MoveNext` method can update the position appropriately. Recursive algorithms, such as that used when walking a binary tree, do not lend themselves to maintaining state information between method calls in an easily accessible manner. For this reason, you will first preprocess the data in the binary tree into a more amenable data structure (a queue) and enumerate this data structure instead. Of course, this deviousness is hidden from the user iterating through the elements of the binary tree!

To create the `TreeEnumerator` class

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **BinaryTree** solution, which is located in the `\Microsoft Press\VCSBS\Chapter 19\BinaryTree` folder in your **Documents** folder. This solution contains a working copy of the `BinaryTree` project you created in Chapter 17. You'll add a new class to this project in which to implement the enumerator for the `BinaryTree` class.
3. In Solution Explorer, select the **BinaryTree** project. Then, open the **Project** menu and select **Add Class** to open the Add New Item – `BinaryTree` dialog.
4. In the middle pane, select the **Class** template, type **`TreeEnumerator.cs`** in the **Name** box, and then select **Add**.

The `TreeEnumerator` class will generate an enumerator for a `Tree<TItem>` object. To ensure that the class is type safe, you must provide a type parameter and implement the `IEnumerator<T>` interface. Also, the type parameter must be a valid type for the `Tree<TItem>` object that the class enumerates, so it must be constrained to implement the `IComparable<TItem>` interface. (The `BinaryTree` class requires that items in the tree provide a means to be compared for sorting purposes.)

5. In the Code and Text Editor window displaying the `TreeEnumerator.cs` file, modify the definition of the `TreeEnumerator` class to satisfy these requirements, as shown in bold in the following example:

```
class TreeEnumerator<TItem> : IEnumerator<TItem> where TItem : IComparable<TItem>
{
}
```



Note Visual Studio will display some warnings and error messages because the `IEnumerator` interface expects you to implement certain methods and properties that you haven't yet added. You'll define these items shortly, so you can ignore these messages.

6. Add the three private variables shown in bold in the following code to the `TreeEnumerator<TItem>` class:

```
class TreeEnumerator<TItem> : IEnumerator<TItem> where TItem : IComparable<TItem>
{
    private Tree<TItem> currentData = null;
    private TItem currentItem = default(TItem);
    private Queue<TItem> enumData = null;
}
```

The `currentData` variable will be used to hold a reference to the tree being enumerated, and the `currentItem` variable will hold the value returned by the `Current` property. You'll populate the `enumData` queue with the values extracted from the nodes in the tree, and the `MoveNext` method will return each item from this queue in turn.



Note The `default` keyword is explained in the sidebar titled “Initializing a variable defined with a type parameter” later in this section.

7. Add a constructor that takes a `Tree<TItem>` parameter called `data` to the `TreeEnumerator<TItem>` class. In the body of the constructor, add the statement shown in bold that initializes the `currentData` variable to `data`:

```
class TreeEnumerator<TItem> : IEnumerator<TItem> where TItem : IComparable<TItem>
{
    ...
    public TreeEnumerator(Tree<TItem> data)
    {
        this.currentData = data;
    }
}
```

8. Add the following private method shown in bold, called `populate`, to the `TreeEnumerator<TItem>` class, immediately after the constructor:

```
class TreeEnumerator<TItem> : IEnumerator<TItem> where TItem : IComparable<TItem>
{
    ...
    public TreeEnumerator(Tree<TItem> data)
    {
        this.currentData = data;
    }

    private void populate(Queue<TItem> enumQueue, Tree<TItem> tree)
    {
        if (tree.LeftTree is not null)
        {
            populate(enumQueue, tree.LeftTree);
        }

        enumQueue.Enqueue(tree.NodeData);

        if (tree.RightTree is not null)
        {
            populate(enumQueue, tree.RightTree);
        }
    }
}
```

```

        populate(enumQueue, tree.RightTree);
    }
}

```

This method walks the binary tree, adding the data it contains to the queue. The algorithm used is similar to that used by the `WalkTree` method in the `Tree<TItem>` class, described in Chapter 17. The main difference is that rather than appending `NodeData` values to a string, the method stores these values in the queue.

9. Return to the definition of the `TreeEnumerator<TItem>` class. In the class declaration, hover over the text `IEnumerator<TItem>`. On the dropdown context menu that appears (with a lightbulb icon), select **Implement all members explicitly**. This generates stubs for the methods in the `IEnumerator<TItem>` interface and the `IEnumerator` interface and adds them to the end of the class. It also generates the `Dispose` method for the `IDisposable` interface.



Note The `IEnumerator<TItem>` interface inherits from the `IEnumerator` and `IDisposable` interfaces, which is why their methods also appear. In fact, the only item that belongs to the `IEnumerator<TItem>` interface is the generic `Current` property. The `MoveNext` and `Reset` methods belong to the nongeneric `IEnumerator` interface. Chapter 14, “Using garbage collection and resource management,” describes the `IDisposable` interface.

10. Examine the code that has been generated. The bodies of the properties and methods contain a default implementation that simply throws a `NotImplementedException` exception. You’ll replace this code in the following steps.
11. Update the body of the `MoveNext` method and replace the `throw new NotImplementedException()` statement with the code shown here in bold:

```

bool IEnumerator.MoveNext()
{
    if (this.enumData is null)
    {
        this.enumData = new Queue<TItem>();
        populate(this.enumData, this.currentData);
    }

    if (this.enumData.Count > 0)
    {
        this.currentItem = this.enumData.Dequeue();
        return true;
    }

    return false;
}

```

The purpose of an enumerator’s `MoveNext` method is twofold. The first time it’s called, it initializes the data used by the enumerator and advances to the first piece of data to be returned. (Before `MoveNext` is called for the first time, the value returned by the `Current` property is undefined and

should result in an exception.) In this case, the initialization process consists of instantiating the queue and then calling the `populate` method to fill the queue with data extracted from the tree.

Subsequent calls to the `MoveNext` method move through data items until there are no more left, dequeuing items until the queue is empty, as in this example. Keep in mind that `MoveNext` doesn't actually return data items; that is the purpose of the `Current` property. All `MoveNext` does is update the internal state in the enumerator (that is, the value of the `currentItem` variable is set to the data item extracted from the queue) for use by the `Current` property, returning `true` if there is a next value and `false` otherwise.

12. Modify the definition of the `get` accessor of the generic `Current` property and replace the expression-bodied member with the following code shown in bold:

```
TItem IEnumerator<TItem>.Current
{
    get
    {
        if (this.enumData is null)
        {
            throw new InvalidOperationException("Use MoveNext before calling Current");
        }
        return this.currentItem;
    }
}
```



Important Be sure to add the code to the correct implementation of the `Current` property. Leave the nongeneric version, `System.Collections.IEnumerator.Current`, with its default implementation, which throws a `NotImplementedException` exception.

The `Current` property examines the `enumData` variable to ensure that `MoveNext` has been called. (This variable will be `null` before the first call to `MoveNext`.) If this is not the case, the property throws an `InvalidOperationException`. This is the conventional mechanism used by .NET applications to indicate that an operation can't be performed in the current state. If `MoveNext` has already been called, it will have updated the `currentItem` variable, so all the `Current` property needs to do is return the value in this variable.

13. Locate the `IDisposable.Dispose` method. Comment out the `throw new NotImplementedException();` statement as shown in bold in the code that follows. The enumerator doesn't use any resources that require explicit disposal, so this method doesn't need to do anything. It must still be present, however. For more information about the `Dispose` method, refer to Chapter 14.

```
void IDisposable.Dispose()
{
    // throw new NotImplementedException();
}
```

14. Build the solution and correct errors if any are reported.

Initializing a variable defined with a type parameter

You should have noticed that the statement that defines and initializes the `currentItem` variable uses the `default` keyword:

```
private TItem currentItem = default(TItem);
```

The `currentItem` variable is defined by using the type parameter `TItem`. When the program is written and compiled, the actual type that will be substituted for `TItem` might not be known; this issue is resolved only when the code is executed. This makes it difficult to specify how the variable should be initialized.

The temptation is to set it to `null`. However, if the type substituted for `TItem` is a value type, this is an illegal assignment. (You cannot set value types to `null`, only reference types.) Similarly, if you set it to `0` with the expectation that the type will be numeric, this will be illegal if the type used is actually a reference type.

There are other possibilities as well. For example, `TItem` could be a `Boolean`. The `default` keyword solves this problem. The value used to initialize the variable will be determined when the statement is executed. If `TItem` is a reference type, `default(TItem)` returns `null`; if `TItem` is numeric, `default(TItem)` returns `0`; if `TItem` is a `Boolean`, `default(TItem)` returns `false`. If `TItem` is a struct, the individual fields in the struct are initialized in the same way. (Reference fields are set to `null`, numeric fields are set to `0`, and `Boolean` fields are set to `false`.)

Implementing the `IEnumerable` interface

In the following exercise, you'll modify the binary tree class to implement the `IEnumerable<T>` interface. The `GetEnumerator` method will return a `TreeEnumerator<TItem>` object.

To implement the `IEnumerable<TItem>` interface in the `Tree<TItem>` class

1. In Solution Explorer, double-click the file **Tree.cs** to display the `Tree<TItem>` class in the Code and Text Editor window.
2. Modify the definition of the `Tree<TItem>` class so that it implements the `IEnumerable<TItem>` interface, as shown in bold in the following code:

```
public class Tree<TItem> : IEnumerable<TItem> where TItem : IComparable<TItem>
```

Notice that constraints are always placed at the end of the class definition.



Note As with the `TreeEnumerator` class, you'll receive some error messages and warnings because the `IEnumerable` interface expects certain methods and properties that you haven't yet defined. You can ignore these messages for the time being.

3. Hover over the `IEnumerable<TItem>` interface in the class definition. On the dropdown context menu that appears, select **Implement all members explicitly**. This generates implementations of the `IEnumerable<TItem>.GetEnumerator` and `IEnumerable.GetEnumerator` methods and adds them to the class. (The nongeneric `IEnumerable` interface method is implemented because the generic `IEnumerable<TItem>` interface inherits from `IEnumerable`.)
4. Locate the generic `IEnumerable<TItem>.GetEnumerator` method near the end of the class. Modify the body of the `GetEnumerator()` method, replacing the existing `throw` statement, as shown in bold in the following example:

```
IEnumerator<TItem> IEnumerable<TItem>.GetEnumerator()  
{  
    return new TreeEnumerator<TItem>(this);  
}
```

The purpose of the `GetEnumerator` method is to construct an enumerator object for iterating through the collection. In this case, all you need to do is build a new `TreeEnumerator<TItem>` object by using the data in the tree.



Note Make sure you leave the nongeneric `GetEnumerator` method with its default implementation.

5. Build the solution. Correct any errors that are reported, and rebuild the solution if necessary.

You'll now test the modified `Tree<TItem>` class by using a `foreach` statement to iterate through a binary tree and display its contents.

To test the enumerator

1. In Solution Explorer, right-click the **BinaryTree** solution, point to **Add**, and then select **New Project**.
2. Select the **Console Application** template, name the project **EnumeratorTest**, and set the location to **\Microsoft Press\VCSBS\Chapter 19\BinaryTree** in your **Documents** folder. Then select the **.NET 6.0 framework** and select **Create**.
3. Right-click the **EnumeratorTest** project in Solution Explorer, and select **Set As StartUp Project**.
4. On the **Project** menu, select **Add Project Reference**.
5. In the left pane of the Reference Manager – EnumeratorTest dialog, expand the **Projects** node and select **Solution**. In the middle pane, select the **BinaryTree** project. Then select **OK**.

The `BinaryTree` assembly appears in the list of dependencies for the `EnumeratorTest` project in Solution Explorer, under the `Projects` folder in the `Dependencies` node.

6. In the Code and Text Editor window displaying the Program class, add the following `using` directive to the top of the file:

```
using BinaryTree;
```

7. Add the statements shown in bold in the following code to the `Main` method to replace the existing code in this method. These statements create and populate a binary tree of integers:

```
static void Main(string[] args)
{
    Tree<int> tree1 = new Tree<int>(10);
    tree1.Insert(5);
    tree1.Insert(11);
    tree1.Insert(5);
    tree1.Insert(-12);
    tree1.Insert(15);
    tree1.Insert(0);
    tree1.Insert(14);
    tree1.Insert(-8);
    tree1.Insert(10);
}
```

8. Add a `foreach` statement, as follows in bold, that enumerates the contents of the tree and displays the results:

```
static void Main(string[] args)
{
    ...
    foreach (int item in tree1)
    {
        Console.WriteLine(item);
    }
}
```

9. On the **Debug** menu, select **Start Without Debugging**.

The program runs and displays the values in the following sequence:

-12, -8, 0, 5, 5, 10, 10, 11, 14, 15

10. Press **Enter** to return to Visual Studio 2022.

Implementing an enumerator by using an iterator

As you can see, the process of making a collection enumerable can become complex and is potentially prone to error. To make life easier, C# provides iterators that can automate much of this process.

An *iterator* is a block of code that yields an ordered sequence of values. An iterator is not actually a member of an enumerable class; rather, it specifies the sequence that an enumerator should use for returning its values. In other words, an iterator is just a description of the enumeration sequence that the C# compiler can use for creating its own enumerator. This concept requires a little thought to understand properly, so consider the following simple example.

A simple iterator

The following `BasicCollection<T>` class illustrates the principles of implementing an iterator. The class uses a `List<T>` object to hold data and provides the `FillList` method to populate this list. Notice also that the `BasicCollection<T>` class implements the `IEnumerable<T>` interface. The `GetEnumerator` method is implemented by using an iterator:

```
using System;
using System.Collections.Generic;
using System.Collections;
class BasicCollection<T> : IEnumerable<T>
{
    private List<T> data = new List<T>();
    public void FillList(params T [] items)
    {
        foreach (var datum in items)
        {
            data.Add(datum);
        }
    }

    IEnumerator<T> IEnumerable<T>.GetEnumerator()
    {
        foreach (var datum in data)
        {
            yield return datum;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Not implemented in this example
        throw new NotImplementedException();
    }
}
```

The `GetEnumerator` method appears to be straightforward, but it warrants closer examination. The first thing to note is that it doesn't appear to return an `IEnumerator<T>` type. Instead, it loops through the items in the `data` array, returning each item in turn. The key point is the use of the `yield` keyword. This keyword indicates the value that should be returned by each iteration. If it helps, you can think of the `yield` statement as calling a temporary halt to the method, passing back a value to the caller. When the caller needs the next value, the `GetEnumerator` method continues at the point at which it left off, looping around and then yielding the next value. Eventually, the data is exhausted, the loop finishes, and the `GetEnumerator` method terminates. At this point, the iteration is complete.

Remember that this is not a normal method in the usual sense. The code in the `GetEnumerator` method defines an *iterator*. The compiler uses this code to generate an implementation of the `IEnumerator<T>` class containing a `Current` method and a `MoveNext` method. This implementation exactly matches the functionality specified by the `GetEnumerator` method. You don't actually get to see this generated code (unless you decompile the assembly containing the compiled code), but that's a small price to pay for the convenience and reduction in code that you need to write. You can invoke

the enumerator generated by the iterator in the usual manner, as shown in the following block of code, which displays the words in the first line of the poem “Jabberwocky” by Lewis Carroll:

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");

foreach (string word in bc)
{
    Console.WriteLine(word);
}
```

This code simply outputs the contents of the `bc` object in this order:

Twas, brillig, and, the, slithy, toves

If you want to provide alternative iteration mechanisms to present the data in a different sequence, you can implement additional properties that implement the `IEnumerable` interface and that use an iterator for returning data. For example, the `Reverse` property of the `BasicCollection<T>` class, shown here, emits the data in the list in reverse order:

```
class BasicCollection<T> : IEnumerable<T>
{
    ...
    public IEnumerable<T> Reverse
    {
        get
        {
            for (int i = data.Count - 1; i >= 0; i--)
            {
                yield return data[i];
            }
        }
    }
}
```

You can invoke this property as follows:

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");
foreach (string word in bc.Reverse)
{
    Console.WriteLine(word);
}
```

This code outputs the contents of the `bc` object in reverse order:

toves, slithy, the, and, brillig, Twas

Defining an enumerator for the `Tree<TItem>` class by using an iterator

In the next exercise, you'll implement the enumerator for the `Tree<TItem>` class by using an iterator. Unlike in the preceding set of exercises, which required the data in the tree to be preprocessed into a queue by the `MoveNext` method, here you can define an iterator that traverses the tree by using a more naturally recursive mechanism, similar to the `WalkTree` method discussed in Chapter 17.

To add an enumerator to the `Tree<TItem>` class

1. Using Visual Studio 2022, open the **BinaryTree** solution, located in the **\Microsoft Press\VCBS\Chapter 19\IteratorBinaryTree** folder in your **Documents** folder. This solution contains another copy of the BinaryTree project you created in Chapter 17.
2. Open the file `Tree.cs` in the Code and Text Editor window. Then modify the definition of the `Tree<TItem>` class so that it implements the `IEnumerable<TItem>` interface, as shown here in bold:

```
public class Tree<TItem> : IEnumerable<TItem> where TItem : IComparable<TItem>
{
    ...
}
```

3. Hover over the `IEnumerable<TItem>` interface in the class definition. On the dropdown context menu that appears, select **Implement all members explicitly** to add the `IEnumerable<TItem>.GetEnumerator` and `IEnumerable.GetEnumerator` methods to the end of the class.
4. Locate the generic `IEnumerable<TItem>.GetEnumerator` method. Replace the contents of the `GetEnumerator` method as shown in bold in the following code:

```
IEnumerator<TItem> IEnumerable<TItem>.GetEnumerator()
{
    if (this.LeftTree is not null)
    {
        foreach (TItem item in this.LeftTree)
        {
            yield return item;
        }
    }

    yield return this.NodeData;

    if (this.RightTree is not null)
    {
        foreach (TItem item in this.RightTree)
        {
            yield return item;
        }
    }
}
```

It might not be obvious at first glance, but this code follows the same recursive algorithm that you used in Chapter 17 to list the contents of a binary tree. If `LeftTree` is not empty, the first `foreach` statement implicitly calls the `GetEnumerator` method (which you are currently defining) over it. This process continues until a node is found that has no left subtree. At this point, the value in the `NodeData` property is yielded, and the right subtree is examined in the same

way. When the right subtree is exhausted, the process unwinds to the parent node, outputting the parent's `NodeData` property and examining the right subtree of the parent. This course of action continues until the entire tree has been enumerated and all the nodes have been output.

To test the new enumerator

1. In Solution Explorer, right-click the **BinaryTree** solution, point to **Add**, and then select **Existing Project**.
2. In the Add Existing Project dialog, move to the folder **\Microsoft Press\VCSBS\Chapter 19\BinaryTree\EnumeratorTest**, select the **EnumeratorTest** project file, and then select **Open**.

This is the project that you created to test the enumerator you developed manually earlier in this chapter.

3. Right-click the **EnumeratorTest** project in Solution Explorer, and then select **Set As StartUp Project**.
4. In Solution Explorer, expand the **Dependencies** folder for the **EnumeratorTest** project, and then expand the **Projects** folder. Right-click the **BinaryTree** reference and then select **Remove**.
5. On the **Project** menu, select **Add Project Reference**.
6. In the left pane of the Reference Manager – EnumeratorTest dialog, expand the **Projects** node and select **Solution**. In the middle pane, select the **BinaryTree** project. Then select **OK**.



Note These three steps ensure that the EnumeratorTest project references the correct version of the BinaryTree assembly. It should use the assembly that implements the enumerator by using the iterator rather than the version created in the previous set of exercises in this chapter.

7. Display the Program.cs file for the EnumeratorTest project in the Code and Text Editor window.
8. Review the `Main` method in the Program.cs file. Recall from testing the earlier enumerator that this method instantiates a `Tree<int>` object, fills it with some data, and then uses a `foreach` statement to display its contents.
9. Build the solution and correct any errors if necessary.
10. On the **Debug** menu, select **Start Without Debugging**.

The program runs and displays the values in the same sequence as before.

-12, -8, 0, 5, 5, 10, 10, 11, 14, 15

11. Press **Enter** and return to Visual Studio 2022.

Summary

In this chapter, you saw how to implement the `IEnumerable<T>` and `IEnumerator<T>` interfaces with a collection class to enable applications to iterate through the items in the collection. You also saw how to implement an enumerator by using an iterator.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 20, “Decoupling application logic and handling events.”
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

| To | Do this |
|--|---|
| Make a collection class enumerable, allowing it to support the foreach construct | Implement the <code>IEnumerable</code> interface and provide a <code>GetEnumerator</code> method that returns an <code>IEnumerator</code> object. For example: <pre>public class Tree<TItem> : IEnumerable<TItem> { ... IEnumerator<TItem> GetEnumerator() { ... } }</pre> |
| Implement an enumerator without using an iterator | Define an enumerator class that implements the <code>IEnumerator</code> interface and that provides the <code>Current</code> property and the <code>MoveNext</code> method (and optionally the <code>Reset</code> method). For example: <pre>public class TreeEnumerator<TItem> : IEnumerator<TItem> { ... TItem Current { get { ... } } bool MoveNext() { ... } }</pre> |
| Define an enumerator by using an iterator | Implement the enumerator to indicate which items should be returned (using the <code>yield</code> statement) and in which order. For example: <pre>IEnumerator<TItem> GetEnumerator() { for (...) { yield return ... } }</pre> |

Decoupling application logic and handling events

After completing this chapter, you will be able to:

- Declare a delegate type to create an abstraction of a method signature, and call a method through a delegate.
- Define a lambda expression to specify the code to be executed by a delegate.
- Enable notifications by using events.
- Understand how the Windows user interface utilizes events to handle user interactions.
- Handle an event by using a delegate.

Many of the examples and exercises in this book have placed great emphasis on the careful definition of classes and structures to enforce encapsulation. In this way, the implementation of the methods in these types can change without unduly affecting the applications that use them. Sometimes, however, it's not possible or desirable to encapsulate the entire functionality of a type. For example, the logic for a method in a class might depend upon which component or application invokes the method, which might need to perform some application or component-specific processing as part of its operation. However, when you build such a class and implement its methods, you might not know which applications and components will use it, and you need to avoid introducing dependencies in your code that might restrict the use of your class. Delegates provide the ideal solution, making it possible for you to fully decouple the application logic in your methods from the applications that invoke them.

Events in C# support a related scenario. Much of the code you've written in the exercises in this book assumes that statements execute sequentially. Although this is the most common case, you'll find that it is sometimes necessary to interrupt the normal flow of execution to perform another, more important task. When that task is complete, the program can continue where it left off.

The classic examples of this style of program are the Universal Windows Platform (UWP) forms that you've been using in the exercises involving graphical applications. A form displays controls such as buttons and boxes. When you click a button or type text in a box, you expect the form to respond immediately—meaning the application must temporarily stop what it's doing to handle your input. This style of operation applies not only to graphical user interfaces (GUIs) but also to any application where an operation must be performed urgently—shutting down the reactor in a nuclear power plant if it is getting too hot, for example.

To handle this kind of processing, the runtime must provide two things: a means of indicating that something urgent has happened, and a way of specifying the code that should be run when the urgent event happens. Events, in conjunction with delegates, provide the infrastructure with which you can implement systems that follow this approach.

Understanding delegates

A *delegate* is a reference to a method. Delegates are so named because they “delegate” processing to the referenced method when they are invoked. A delegate is a very simple concept with extraordinarily powerful implications. Let me explain.

Typically, when you write a statement that invokes a method, you specify the name of the method (and possibly the object or structure to which the method belongs). It’s clear from your code exactly which method you’re running and when you are running it. Look at the following simple example that calls the `performCalculation` method of a `Processor` object (what this method does or how the `Processor` class is defined is immaterial for this discussion):

```
Processor p = new Processor();  
p.performCalculation();
```

You can assign a reference to a method to a delegate in much the same way that you can assign an `int` value to an `int` variable. The next example creates a delegate named `performCalculationDelegate` that references the `performCalculation` method of the `Processor` object. I have deliberately omitted some elements of the statement that declares the delegate because it’s more important to understand the concept than to worry about the syntax. (You’ll see the full syntax shortly.)

```
Processor p = new Processor();  
delegate ... performCalculationDelegate ...;  
performCalculationDelegate = p.performCalculation;
```

Keep in mind that the statement that assigns the method reference to the delegate does not run the method at that point; there are no parentheses after the method name, and you do not specify any parameters (if the method takes them). This is just an assignment statement.

Having stored a reference to the `performCalculation` method of the `Processor` object in the delegate, the application can subsequently invoke the method through the delegate, like this:

```
performCalculationDelegate();
```

This looks like an ordinary method call. If you didn’t know otherwise, it looks like you might actually be running a method named `performCalculationDelegate`. However, the common language runtime (CLR) knows that this is a delegate, so it retrieves the method that the delegate references and runs that instead. Later, you can change the method to which a delegate refers, so a statement that calls a delegate might actually run a different method each time it executes. Additionally, a delegate can reference more than one method at a time (think of it as a collection of method references). When you invoke a delegate, all the methods it refers to will run.



Note If you're familiar with C++, a delegate is similar to a function pointer. However, unlike function pointers, delegates are completely type safe. You can make a delegate refer only to a method that matches the signature of the delegate, and you cannot invoke a delegate that does not refer to a valid method.

Examples of delegates in the .NET class library

The Microsoft .NET class library makes extensive use of delegates for many of its types. Two examples are in Chapter 18, "Using collections": the `Find` method and the `Exists` method of the `List<T>` class. As you might recall, these methods search through a `List<T>` collection, either returning a matching item or testing for the existence of a matching item. When the designers of the `List<T>` class were implementing it, they had absolutely no idea about what should actually constitute a match in your application code, so they let you define that by providing your own code in the form of a predicate. A predicate is really just a delegate that happens to return a Boolean value.

The following code should help remind you how to use the `Find` method (shown in bold):

```
struct Person
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
...
List<Person> personnel = new List<Person>()
{
    new Person() { ID = 1, Name = "John", Age = 53 },
    new Person() { ID = 2, Name = "Sid", Age = 28 },
    new Person() { ID = 3, Name = "Fred", Age = 34 },
    new Person() { ID = 4, Name = "Paul", Age = 22 }
};
...
// Find the member of the list that has an ID of 3
Person match = personnel.Find(p => p.ID == 3);
```

Other examples of methods exposed by the `List<T>` class that use delegates to perform their operations are `Average`, `Max`, `Min`, `Count`, and `Sum`. These methods take a `Func` delegate as the parameter. A `Func` delegate refers to a method that returns a value (a function). In the following examples, the `Average` method is used to calculate the average age of items in the `personnel` collection (the `Func<T>` delegate simply returns the value in the `Age` field of each item in the collection), the `Max` method is used to determine the item with the highest ID, and the `Count` method calculates how many items have an `Age` value between 30 and 39, inclusively.

```
double averageAge = personnel.Average(p => p.Age);
Console.WriteLine($"Average age is {averageAge}");
...
```

```
int id = personnel.Max(p => p.ID);
Console.WriteLine($"Person with highest ID is {id}");
...
int thirties = personnel.Count(p => p.Age >= 30 && p.Age <= 39);
Console.WriteLine($"Number of personnel in their thirties is {thirties}");
```

This code generates the following output:

```
Average age is 34.25
Person with highest ID is 4
Number of personnel in their thirties is 1
```

You'll meet many examples of these and other delegate types used by the .NET class library throughout the remainder of this book. You can also define your own delegates. The best way to fully understand how and when you might want to do this is to see them in action, so next, you'll work through an example.

The `Func<T, ...>` and `Action<T, ...>` delegate types

The parameter taken by the `Average`, `Max`, `Count`, and other methods of the `List<T>` class is actually a generic `Func<T, TResult>` delegate. The type parameters refer to the type of the parameter passed to the delegate and the type of the return value. For the `Average`, `Max`, and `Count` methods of the `List<Person>` class shown in the text, the first type parameter `T` is the type of data in the list (the `Person` struct), whereas the `TResult` type parameter is determined by the context in which the delegate is used. In the following example, the type of `TResult` is `int` because the value returned by the `Count` method should be an integer:

```
int thirties = personnel.Count(p => p.Age >= 30 && p.Age <= 39);
```

So, in this example, the type of the delegate expected by the `Count` method is `Func<Person, int>`.

This point might seem somewhat academic because the compiler automatically generates the delegate based on the type of the `List<T>`, but it's worth familiarizing yourself with this idiom as it occurs time and again throughout the .NET class library. In fact, the `System` namespace defines an entire family of `Func` delegate types, from `Func<TResult>` for functions that return a result without taking any parameters to `Func<T1, T2, T3, T4, ..., T16, TResult>` for functions that take 16 parameters. If you find yourself in a situation in which you're creating your own delegate type that matches this pattern, you should consider using an appropriate `Func` delegate type instead. You'll meet the `Func` delegate types again in Chapter 21, "Querying in-memory data by using query expressions."

Alongside `Func`, the `System` namespace also defines a series of `Action` delegate types. An `Action` delegate is used to reference a method that performs an action instead of returning a value (a void method). Again, a family of `Action` delegate types is available ranging from `Action<T>` (specifying a delegate that takes a single parameter) to `Action<T1, T2, T3, T4, ..., T16>`.

The automated factory scenario

Suppose you're writing the control systems for an automated factory. The factory contains numerous different machines, each performing distinct tasks in the production of the articles manufactured by the factory—shaping and folding metal sheets, welding sheets together, painting sheets, and so on. Each machine was built and installed by a specialist vendor. The machines are all controlled by a computer, and each vendor has provided a set of functions that you can use to control its machine. Your task is to integrate the different systems used by the machines into a single control program. One aspect you've decided to concentrate on is to provide a means of shutting down all the machines—quickly, if needed!

Each machine has its own unique computer-controlled process (and functions) for shutting down safely, as summarized here:

```
StopFolding(); // Folding and shaping machine
FinishWelding(); // Welding machine
PaintOff(); // Painting machine
```

Implementing the factory control system without using delegates

A simple approach to implementing the shutdown functionality in the control program is as follows:

```
class Controller
{
    // Fields representing the different machines
    private FoldingMachine folder;
    private WeldingMachine welder;
    private PaintingMachine painter;
    ...
    public void ShutDown()
    {
        folder.StopFolding();
        welder.FinishWelding();
        painter.PaintOff();
    }
    ...
}
```

Although this approach works, it's not very extensible or flexible. If the factory buys a new machine, you must modify this code because the `Controller` class and the code for managing the machines is tightly coupled.

Implementing the factory by using a delegate

Although the names of each method are different, they all have the same "shape": they take no parameters, and they don't return a value. (You'll consider what happens if this isn't the case later, so bear with me.) The general format of each method is this:

```
void methodName();
```

This is where a delegate can be useful. You can use a delegate that matches this shape to refer to any of the machinery shutdown methods. You declare a delegate like this:

```
delegate void stopMachineryDelegate();
```

Note the following points:

- You use the `delegate` keyword.
- You specify the return type (`void` in this example), a name for the delegate (`stopMachineryDelegate`), and any parameters (there are none in this case).

After you've declared the delegate, you can create an instance of it and use the `+=` compound assignment operator to make it refer to a matching method. You do this in the constructor of the controller class, like this:

```
class Controller
{
    delegate void stopMachineryDelegate(); // the delegate type
    private stopMachineryDelegate stopMachinery; // an instance of the delegate
    ...
    public Controller()
    {
        this.stopMachinery += folder.StopFolding;
    }
    ...
}
```

It takes a bit of study to get used to this syntax. You *add* the method to the delegate—remember, you are not actually calling the method at this point. The `+` operator is overloaded to have this new meaning when used with delegates. (You'll learn more about operator overloading in Chapter 22, "Operator overloading.") Notice that you simply specify the method name and do not include any parentheses or parameters.

It's safe to use the `+=` operator on an uninitialized delegate. It will be initialized automatically. Alternatively, you can use the `new` keyword to initialize a delegate explicitly with a single specific method, like this:

```
this.stopMachinery = new stopMachineryDelegate(folder.StopFolding);
```

You can call the method by invoking the delegate, like this:

```
public void ShutDown()
{
    this.stopMachinery();
    ...
}
```

You use the same syntax to invoke a delegate as you use to call a method. If the method to which the delegate refers takes any parameters, you should specify them at this time between the parentheses.



Note If you attempt to invoke a delegate that is uninitialized and does not refer to any methods, you'll get a `NullReferenceException` exception.

An important advantage of using a delegate is that it can refer to more than one method at the same time. You simply use the `+=` operator to add methods to the delegate, like this:

```
public Controller()
{
    this.stopMachinery += folder.StopFolding;
    this.stopMachinery += welder.FinishWelding;
    this.stopMachinery += painter.PaintOff;
}
```

Invoking `this.stopMachinery()` in the `Shutdown` method of the `Controller` class automatically calls each of the methods in turn. The `Shutdown` method doesn't need to know how many machines there are or what the method names are.

You can remove a method from a delegate by using the `-` compound assignment operator, as demonstrated here:

```
this.stopMachinery -= folder.StopFolding;
```

The current scheme adds the machine methods to the delegate in the `Controller` constructor. To make the `Controller` class totally independent of the various machines, you need to make the `stopMachineryDelegate` type public and supply a means of enabling classes outside `Controller` to add methods to the delegate. You have several options:

- Make the `stopMachinery` delegate variable public:

```
public stopMachineryDelegate stopMachinery;
```

- Keep the `stopMachinery` delegate variable private, but create a read/write property to provide access to it:

```
private stopMachineryDelegate stopMachinery;
...
public stopMachineryDelegate StopMachinery
{
    get => this.stopMachinery;
    set => this.stopMachinery = value;
}
```

- Provide complete encapsulation by implementing separate `Add` and `Remove` methods. The `Add` method takes a method as a parameter and adds it to the delegate, whereas the `Remove` method removes the specified method from the delegate. (Notice that you specify a method as a parameter by using a delegate type.)

```
public void Add(stopMachineryDelegate stopMethod) => this.stopMachinery += stopMethod;

public void Remove(stopMachineryDelegate stopMethod) => this.stopMachinery -= stopMethod;
```

An object-oriented purist would probably opt for the Add/Remove approach. However, the other approaches are viable alternatives and are frequently used, which is why they are shown here.

Whichever technique you choose, you should remove the code that adds the machine methods to the delegate from the `Controller` constructor. You can then instantiate a `Controller` and objects representing the other machines like this (this example uses the Add/Remove approach):

```
Controller control = new Controller();
FoldingMachine folder = new FoldingMachine();
WeldingMachine welder = new WeldingMachine();
PaintingMachine painter = new PaintingMachine();
...
control.Add(folder.StopFolding);
control.Add(welder.FinishWelding);
control.Add(painter.PaintOff);
...
control.ShutDown();
...
```

Declaring and using delegates

In the following exercises, you'll complete an application that forms part of a system for a company called Wide World Importers, which imports and sells building materials and tools. The application that you'll work on gives customers the ability to browse the items that Wide World Importers currently has in stock and place orders for these items. The application contains a form that displays the goods currently available, together with a pane that lists the items that a customer has selected. When the customer wants to place an order, she can select the Checkout button on the form. The order is then processed, and the pane is cleared.

Currently, when the customer places an order, several actions occur:

- Payment is requested from the customer.
- The items in the order are examined, and if any of them are age restricted (such as the power tools), details of the order are audited and tracked.
- A dispatch note is generated for shipping purposes. This dispatch note contains a summary of the order.

The logic for the auditing and shipping processes is independent of the checkout logic, although the order in which these processes occur is immaterial. Furthermore, either of these elements might be amended in the future, and additional processing might be required by the checkout operation as business circumstances or regulatory requirements change. Therefore, it is desirable to decouple the payment and checkout logic from the auditing and shipping processes to make maintenance and upgrades easier.

You'll start by examining the application to see how it currently fails to fulfill this objective. You will then modify the structure of the application to remove the dependencies between the checkout logic and the auditing and shipping logic.

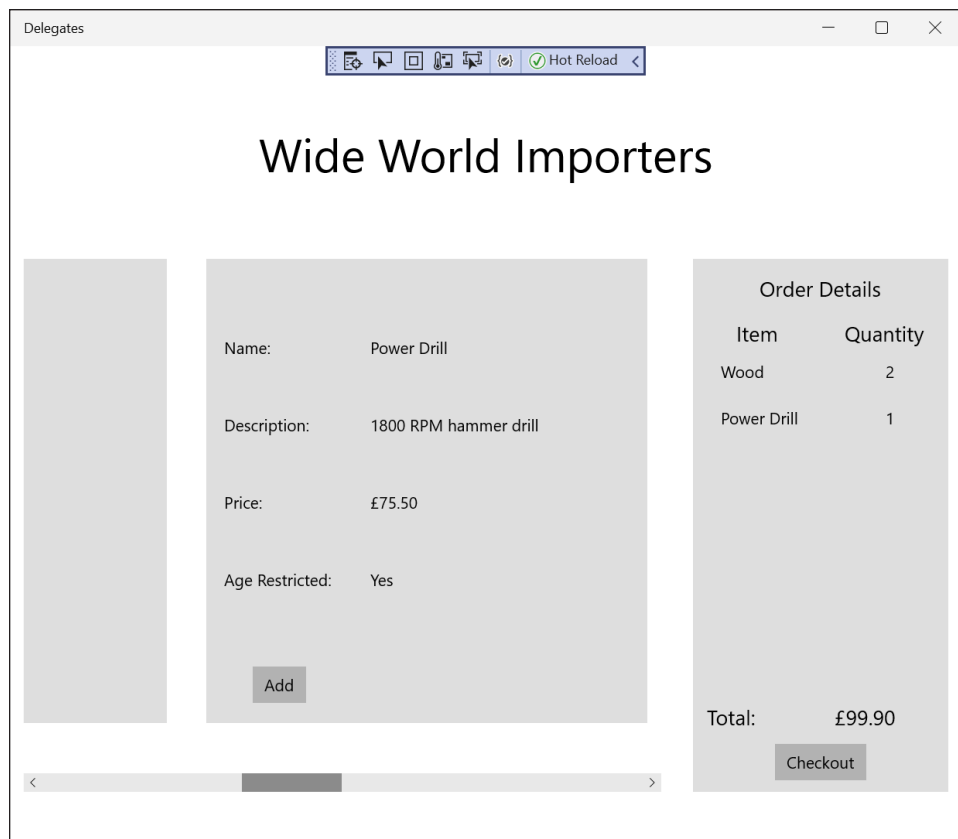
To examine the Wide World Importers application

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **Delegates** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 20\Delegates** folder in your **Documents** folder.
3. On the **Debug** menu, select **Start Debugging**.

The project builds and runs. A form appears displaying the items available, together with a panel showing the details of the order (it is empty initially). The app displays the items in a `GridView` control that scrolls horizontally.

4. Select one or more items and then select **Add** to include them in the shopping basket. Be sure to select at least one age-restricted item.

When you add an item, it appears in the Order Details pane on the right. If you add the same item more than once, the quantity is incremented each time. (This version of the application does not implement functionality to remove items from the basket.) The currency used by the application depends on your locale. I'm based in the UK, so values displayed in the following image are in sterling. If you're in the United States, however, you will see values in dollars.



5. In the Order Details pane, select **Checkout**.

A message appears indicating that the order has been placed. The order is given a unique ID, and this ID is displayed together with the value of the order.



6. Select **Close** to dismiss the message. Then return to the Visual Studio 2022 environment and stop debugging.

7. In Solution Explorer, expand the **Delegates** project node, and then open the **Package.appxmanifest** file.

The package manifest editor appears.

8. In the package manifest editor, select the **Packaging** tab.

Note the value in the Package Name field. It takes the form of a globally unique identifier (GUID).

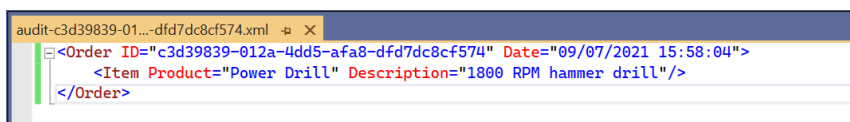
9. Using File Explorer, browse to **%USERPROFILE%\AppData\Local\Packages\yyy\LocalState**, where **yyy** is an identifier value that begins with the GUID you noted in the previous step. This is the local folder for the Wide World Importers application. You should see two files:

- **audit-*nnnnnn*.xml (where *nnnnnn* is the ID of the order displayed earlier)** This file was generated by the app's auditing component.
- **dispatch-*nnnnnn*.txt (again, where *nnnnnn* is the ID of the order displayed earlier)** This file is the dispatch note generated by the shipping component.



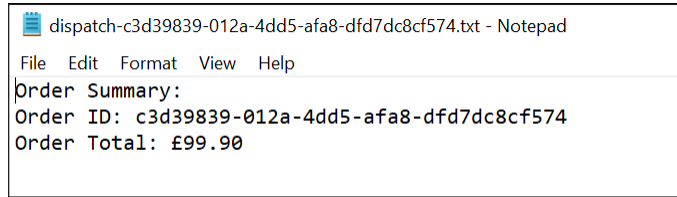
Note If there is no `audit-nnnnnn.xml` file, then you didn't select any age-restricted items when you placed the order. In this case, switch back to the application and create a new order that includes one or more age-restricted items.

10. Open the `audit-nnnnnn.xml` file in Visual Studio. This file contains a list of the age-restricted items in the order together with the order number and date. The file is in XML format and should look similar to this:



```
<Order ID="c3d39839-012a-4dd5-afa8-dfd7dc8cf574" Date="09/07/2021 15:58:04">
  <Item Product="Power Drill" Description="1800 RPM hammer drill"/>
</Order>
```

11. Examine the list and then close the file in Visual Studio.
12. Open the **dispatch-*nnnnnn*.txt** file in Notepad. This file contains a summary of the order, listing the order ID and the value. This is an ordinary text file and should look similar to this:



The screenshot shows a Notepad window titled "dispatch-c3d39839-012a-4dd5-afa8-dfd7dc8cf574.txt - Notepad". The menu bar includes File, Edit, Format, View, and Help. The text content is as follows:

```
Order Summary:
Order ID: c3d39839-012a-4dd5-afa8-dfd7dc8cf574
Order Total: £99.90
```

13. Close Notepad, return to Visual Studio 2022, and stop debugging.
14. In Visual Studio, notice that the solution consists of the following projects:
 - **Delegates** This project contains the application itself. The MainPage.xaml file defines the user interface, and the application logic is contained in the MainPage.xaml.cs file.
 - **AuditService** This project contains the component that implements the auditing process. It's packaged as a class library and contains a single class named Auditor. This class exposes a single public method, AuditOrder, that examines an order and generates the audit-*nnnnnn*.xml file if the order contains any age-restricted items.
 - **DeliveryService** This project contains the component that performs the shipping logic, packaged as a class library. The shipping functionality is contained in the Shipper class, and it provides a public method named ShipOrder that handles the shipping process and also generates the dispatch note.



Note You're welcome to examine the code in the Auditor and Shipper classes, but it's not necessary to fully understand the inner workings of these components in this application.

- **DataTypes** This project contains the data types used by the other projects. The Product class defines the details of the products displayed by the application, and the data for the products is held in the ProductsDataSource class. (The application currently uses a small hard-coded set of products. In a production system, this information would be retrieved from a database or web service.) The Order and OrderItem classes implement the structure of an order; each order contains one or more order items.

15. In the Delegates project, display the MainPage.xaml.cs file in the Code and Text Editor window and examine the private fields and MainPage constructor in this file. The important elements look like this:

```
...
private Auditor auditor = null;
private Shipper shipper = null;

public MainPage()
{
    ...
    this.auditor = new Auditor();
    this.shipper = new Shipper();
}
```

The `auditor` and `shipper` fields contain references to instances of the `Auditor` and `Shipper` classes, and the constructor instantiates these objects.

16. Locate the `CheckoutButtonClicked` method. This method runs when the user selects the Checkout button to place an order. The first few lines look like this:

```
private void CheckoutButtonClicked(object sender, RoutedEventArgs e)
{
    try
    {
        // Perform the checkout processing
        if (this.requestPayment())
        {
            this.auditor.AuditOrder(this.order);
            this.shipper.ShipOrder(this.order);
        }
        ...
    }
    ...
}
```

This method implements the checkout processing. It requests payment from the customer and then invokes the `AuditOrder` method of the `auditor` object followed by the `ShipOrder` method of the `shipper` object. Any additional business logic required in the future can be added here.

The remainder of the code in this method (after the `if` statement) is concerned with managing the user interface: displaying the message box to the user and clearing out the Order Details pane.



Note For simplicity, the `requestPayment` method in this application currently just returns `true` to indicate that payment has been received. In the real world, this method would perform the complete payment processing and verification.

Although the application operates as advertised, the `Auditor` and `Shipper` components are tightly integrated into the checkout processing. If these components change, the application will need to be updated. Similarly, if you need to incorporate additional logic into the checkout process, possibly performed by using other components, you'll need to amend this part of the application.

In the next exercise, you'll see how you can decouple the business processing for the checkout operation from the application. The checkout processing will still need to invoke the `Auditor` and `Shipper` components, but it must be extensible enough to allow additional components to be easily incorporated. You will achieve this by creating a component called `CheckoutController`. The `CheckoutController` component will implement the business logic for the checkout process and expose a delegate that enables an application to specify which components and methods should be included within this process. The `CheckoutController` component will invoke these methods by using the delegate.

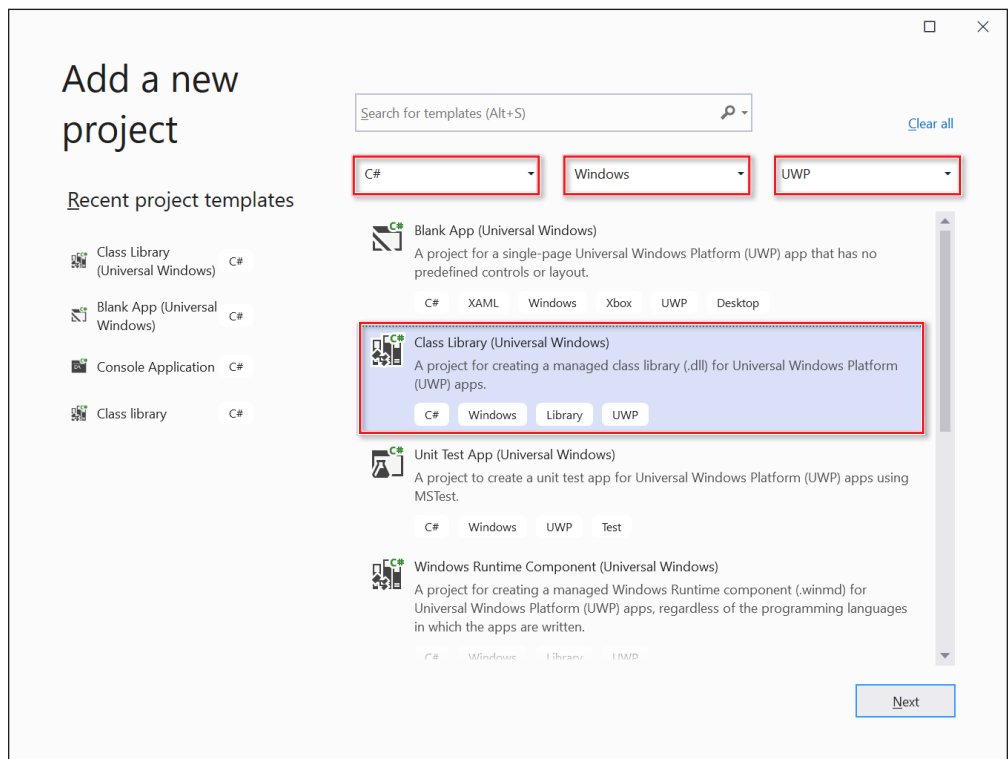
To create the `CheckoutController` component

1. In Solution Explorer, right-click the **Delegates** solution, point to **Add**, and then select **New Project**.



Note Make sure you select the **Delegates** solution and not the **Delegates** project.

2. In the Add a new project dialog, set the template selectors to **C#**, **Windows**, and **UWP**. Select the **Class Library (Universal Windows)** template, and then select **Next**.



3. In the Configure your new project dialog, in the **Project name** box, type **CheckoutService**. Set the **Location** to the **\Microsoft Press\VCSBS\Chapter 20\Delegates** folder under your **Documents** folder, and then select **Create**.
4. In the New Universal Windows Platform Project dialog, accept the default values for **Target version** and **Minimum version**, and then select **OK**.
5. In Solution Explorer, expand the **CheckoutService** project, right-click the file **Class1.cs**, and then select **Rename**. Change the name of the file to **CheckoutController.cs** and then press **Enter**. Allow Visual Studio to rename all references to **Class1** as **CheckoutController** when prompted.
6. Right-click the **References** node in the **CheckoutService** project, and then select **Add Reference**.
7. In the left pane of the Reference Manager – CheckoutService dialog, select **Solution**. In the middle pane, select the **DataTypes** project. Then select **OK**.

The **CheckoutController** class will use the **Order** class defined in the **DataTypes** project.

8. In the Code and Text Editor window displaying the **CheckoutController.cs** file, add the following **using** directive to the list at the top of the file:

```
using DataTypes;
```

9. Add a public delegate type called **CheckoutDelegate** to the **CheckoutController** class, as shown in the following in bold:

```
public class CheckoutController
{
    public delegate void CheckoutDelegate(Order order);
}
```

You can use this delegate type to reference methods that take an **Order** parameter and that do not return a result. This just happens to match the shape of the **AuditOrder** and **ShipOrder** methods of the **Auditor** and **Shipper** classes.

10. Add a public delegate called **CheckoutProcessing** based on this delegate type, like this:

```
public class CheckoutController
{
    public delegate void CheckoutDelegate(Order order);
    public CheckoutDelegate CheckoutProcessing = null;
}
```

11. Display the **MainPage.xaml.cs** file of the **Delegates** project in the Code and Text Editor window and locate the **requestPayment** method (it's at the end of the file). Cut this method from the **MainPage** class.

12. Return to the CheckoutController.cs file and paste the requestPayment method into the CheckoutController class, as shown in bold in the following:

```
public class CheckoutController
{
    public delegate void CheckoutDelegate(Order order);
    public CheckoutDelegate CheckoutProcessing = null;

    private bool requestPayment()
    {
        // Payment processing goes here
        // Payment logic is not implemented in this example
        // Simply return true to indicate payment has been received
        return true;
    }
}
```

13. Add the StartCheckoutProcessing method shown here in bold to the CheckoutController class:

```
public class CheckoutController
{
    public delegate void CheckoutDelegate(Order order);
    public CheckoutDelegate CheckoutProcessing = null;

    private bool requestPayment()
    {
        ...
    }

    public void StartCheckoutProcessing(Order order)
    {
        // Perform the checkout processing
        if (this.requestPayment())
        {
            if (this.CheckoutProcessing is not null)
            {
                this.CheckoutProcessing(order);
            }
        }
    }
}
```

This method provides the checkout functionality previously implemented by the CheckoutButtonClicked method of the MainPage class. It requests payment and then examines the CheckoutProcessing delegate. If this delegate is not null (it refers to one or more methods), it invokes the delegate. Any methods referenced by this delegate will run at this point.

14. In Solution Explorer, in the Delegates project, right-click the **References** node and then select **Add Reference**.
15. In the left pane of the Reference Manager – Delegates dialog, select **Solution**. In the middle pane, select the **CheckoutService** project. (Leave the other projects selected as well.) Then select **OK**.

16. Return to the MainPage.xaml.cs file of the Delegates project and add the following using directive to the list at the top of the file:

using CheckoutService;

17. Add a private variable named `checkoutController` of type `CheckoutController` to the `MainPage` class, and initialize it to `null`, as shown in bold in the following:

```
public ... class MainPage : ...
{
    ...
    private Auditor auditor = null;
    private Shipper shipper = null;
    private CheckoutController checkoutController = null;
    ...
}
```

18. Locate the `MainPage` constructor. After the statements that create the `Auditor` and `Shipper` components, instantiate the `CheckoutController` component, as follows in bold:

```
public MainPage()
{
    ...
    this.auditor = new Auditor();
    this.shipper = new Shipper();
    this.checkoutController = new CheckoutController();
}
```

19. After the statement you just entered, add the following statements shown in bold to the constructor:

```
public MainPage()
{
    ...
    this.checkoutController = new CheckoutController();
    this.checkoutController.CheckoutProcessing += this.auditor.AuditOrder;
    this.checkoutController.CheckoutProcessing += this.shipper.ShipOrder;
}
```

This code adds references to the `AuditOrder` and `ShipOrder` methods of the `Auditor` and `Shipper` objects to the `CheckoutProcessing` delegate of the `CheckoutController` object.

20. Find the `CheckoutButtonClicked` method. In the try block, replace the code that performs the checkout processing (the if statement block) with the statement shown here in bold:

```
private void CheckoutButtonClicked(object sender, RoutedEventArgs e)
{
    try
    {
        // Perform the checkout processing
        this.checkoutController.StartCheckoutProcessing(this.order);

        // Display a summary of the order
        ...
    }
    ...
}
```

You have now decoupled the checkout logic from the components that this checkout process uses. The business logic in the `MainPage` class specifies which components the `CheckoutController` should use.

To test the application

1. On the **Debug** menu, select **Start Debugging** to build and run the application.
2. When the Wide World Importers form appears, select some items (include at least one age-restricted item), and then select **Checkout**.
3. When the Order Placed message appears, make a note of the order number, and then select **Close**.
4. Switch to File Explorer and move to the `%USERPROFILE%\AppData\Local\Packages\yyy\LocalState` folder, where `yyy` is an identifier value that begins with the GUID for the application that you noted previously.
5. Verify that a new `audit-nnnnnn.xml` file and `dispatch-nnnnnn.txt` file have been created, where `nnnnnn` is the number that identifies the new order. Examine these files and verify that they contain the details of the order.
6. Return to Visual Studio 2022 and stop debugging.

Lambda expressions and delegates

All the examples of adding a method to a delegate that you've seen so far use the method's name. For example, in the automated factory scenario described earlier, you add the `StopFolding` method of the `folder` object to the `stopMachinery` delegate like this:

```
this.stopMachinery += folder.StopFolding;
```

This approach is very useful if there's a convenient method that matches the signature of the delegate. But what if the `StopFolding` method actually had the following signature:

```
void StopFolding(int shutDownTime); // Shut down in the specified number of seconds
```

This signature is now different from that of the `FinishWelding` and `PaintOff` methods, and therefore you cannot use the same delegate to handle all three methods. What do you do? One way around this problem is to create another method that calls `StopFolding`, but that takes no parameters itself, like this:

```
void FinishFolding()
{
    folder.StopFolding(0); // Shut down immediately
}
```


You can then add the `FinishFolding` method to the `stopMachinery` delegate in place of the `StopFolding` method, using the same syntax as before:

```
this.stopMachinery += folder.FinishFolding;
```

When the `stopMachinery` delegate is invoked, it calls `FinishFolding`, which in turn calls the `StopFolding` method, passing in the parameter 0.



Note The `FinishFolding` method is a classic example of an adapter—a method that converts (or adapts) a method to give it a different signature. This pattern is very common and is one of the sets of patterns documented in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1994).

In many cases, adapter methods such as this are small, and it's easy to lose them in a sea of methods, especially in a large class. Furthermore, the method is unlikely to be called except for its use in adapting the `StopFolding` method for use by the delegate. C# provides lambda expressions for situations such as this. Lambda expressions were described in Chapter 18, and there are more examples of them earlier in this chapter. In the factory scenario, you can use the following lambda expression:

```
this.stopMachinery += () => folder.StopFolding(0);
```

When you invoke the `stopMachinery` delegate, it runs the code defined by the lambda expression, which, in turn, calls the `StopFolding` method with the appropriate parameter.

Enabling notifications by using events

You've now seen how to declare a delegate type, call a delegate, and create delegate instances. However, this is only half the story. Although you can invoke any number of methods indirectly by using delegates, you still must invoke the delegate explicitly.

In many cases, it would be useful to have the delegate run automatically when something significant happens. For example, in the automated factory scenario, it could be vital to be able to invoke the `stopMachinery` delegate and halt the equipment if the system detects that a machine is overheating.

The .NET libraries provide *events*, which you can use to define and trap significant actions and arrange for a delegate to be called to handle the situation. Many classes in .NET expose events. Most of the controls that you can place on a form in a UWP app, and the `Windows` class itself, use events to run code when, for example, the user selects a button or types something in a box. You can also declare your own events.

Declaring an event

You declare an event in a class intended to act as an event source. An *event source* is usually a class that monitors its environment and raises an event when something significant happens.

In the automated factory, an event source could be a class that monitors the temperature of each machine. The temperature-monitoring class would raise a “machine overheating” event if it detects that a machine has exceeded its thermal radiation boundary (that is, it has become too hot).

An event maintains a list of methods to call when it is raised. These methods are sometimes referred to as *subscribers*. Returning to the automatic factory example, these methods should be prepared to handle the “machine overheating” event and take the necessary corrective action: shut down the machines.

You declare an event similarly to how you declare a field. However, because events are intended to be used with delegates, the type of an event must be a delegate, and you must prefix the declaration with the event keyword. Use the following syntax to declare an event:

```
event delegateTypeName eventName
```

As an example, here’s the `StopMachineryDelegate` delegate from the automated factory scenario. It has been relocated to a class named `TemperatureMonitor`, which provides an interface to the various electronic probes monitoring the temperature of the equipment. (This is a more logical place for the event than the `Controller` class.)

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    ...
}
```

You can define the `MachineOverheating` event, which will invoke the `stopMachineryDelegate`, like this:

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    public event StopMachineryDelegate MachineOverheating;
    ...
}
```

The logic (not shown) in the `TemperatureMonitor` class raises the `MachineOverheating` event as necessary. (You’ll see how to raise an event in an upcoming section.) Also, you add methods to an event (a process known as *subscribing* to the event) rather than add them to the delegate on which the event is based. You’ll look at this aspect of events next.

Subscribing to an event

Like delegates, events come ready-made with a `+=` operator, and you use this operator to subscribe to an event. In the automated factory example, the software controlling each machine can arrange for the shutdown methods to be called when the `MachineOverheating` event is raised, like this:

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    public event StopMachineryDelegate MachineOverheating;
    ...
}
...
TemperatureMonitor tempMonitor = new TemperatureMonitor();
...
tempMonitor.MachineOverheating += (() => { folder.StopFolding(0); });
tempMonitor.MachineOverheating += welder.FinishWelding;
tempMonitor.MachineOverheating += painter.PaintOff;
```

Notice that the syntax is the same as for adding a method to a delegate. You can even subscribe by using a lambda expression. When the `tempMonitor.MachineOverheating` event runs, it will call all the subscribing methods and shut down the machines.

Unsubscribing from an event

Knowing that you use the `+=` operator to attach a delegate to an event, you can probably guess that you use the `-=` operator to detach a delegate from an event. Calling the `-=` operator removes the method from the event's internal delegate collection. This action is often referred to as *unsubscribing* from the event.

Raising an event

You can raise an event by calling it like a method. When you raise an event, all the attached delegates are called in sequence. For example, here's the `TemperatureMonitor` class with a private `Notify` method that raises the `MachineOverheating` event:

```
class TemperatureMonitor
{
    public delegate void StopMachineryDelegate();
    public event StopMachineryDelegate MachineOverheating;
    ...
    private void Notify()
    {
        if (this.MachineOverheating is not null)
        {
            this.MachineOverheating();
        }
    }
    ...
}
```

This is a common idiom. The `null` check is necessary because an event field is implicitly `null` and only becomes nonnull when a method subscribes to it by using the `+=` operator. If you try to raise a `null` event, you'll get a `NullReferenceException` exception. If the delegate defining the event expects any parameters, the appropriate arguments must be provided when you raise the event. You'll see some examples of this later.



Important Events have a very useful built-in security feature. A public event (such as `MachineOverheating`) can be raised only by methods in the class that define it (the `TemperatureMonitor` class). Any attempt to raise the event outside the class results in a compiler error.

Understanding user-interface events

As mentioned, .NET classes and controls used to build GUIs employ events extensively. For example, the `Button` class derives from the `ButtonBase` class, inheriting a public event called `Click` of type `RoutedEventHandler`. The `RoutedEventHandler` delegate expects two parameters: a reference to the object that caused the event to be raised, and a `RoutedEventArgs` object that contains additional information about the event:

```
public delegate void RoutedEventHandler(Object sender, RoutedEventArgs e);
```

The `Button` class looks like this:

```
public class ButtonBase:
...
{
    public event RoutedEventHandler Click;
    ...
}

public class Button: ButtonBase
{
    ...
}
```

The `Button` class automatically raises the `Click` event when you select the button on the screen. This arrangement makes it easy to create a delegate for a chosen method and attach that delegate to the required event.

The following example shows the code for a UWP form that contains a button named *okay*, and the code to connect the `Click` event of the *okay* button to the `okayClick` method:

```
partial class MainPage :
    global::Windows.UI.Xaml.Controls.Page,
    global::Windows.UI.Xaml.Markup.IComponentConnector,
    global::Windows.UI.Xaml.Markup.IComponentConnector2
{
    ...
    public void Connect(int connectionId, object target)
```

```

{
    switch(connectionId)
    {
        case 1:
        {
            this.okay = (global::Windows.UI.Xaml.Controls.Button)(target);
            ...
            ((global::Windows.UI.Xaml.Controls.Button)this.okay).Click += this.okayClick;
            ...
        }
        break;
        default:
        {
            break;
        }
    }
    this._contentLoaded = true;
}
...
}

```

This code is usually hidden from you. When you use the Design View window in Visual Studio 2022 and set the `Click` property of the okay button to `okayClick` in the Extensible Application Markup Language (XAML) description of the form, Visual Studio 2022 generates this code for you. All you have to do is write your application logic in the event-handling method, `okayClick`, in the part of the code you have access to, which is the `MainPage.xaml.cs` file in this case:

```

public sealed partial class MainPage : Page
{
    ...
    private void okayClick(object sender, RoutedEventArgs e)
    {
        // your code to handle the Click event
    }
}

```

The events that the various GUI controls generate always follow the same pattern. The events are of a delegate type whose signature has a `void` return type and two arguments. The first argument is always the sender (the source) of the event, and the second argument is always an `EventArgs` argument (or a class derived from `EventArgs`).

With the sender argument, you can reuse a single method for multiple events. The delegated method can examine the sender argument and respond accordingly. For example, you can use the same method to subscribe to the `Click` event for two buttons. (You add the same method to two different events.) When the event is raised, the code in the method can examine the sender argument to ascertain which button was selected.

Using events

In the previous exercise, you amended the Wide World Importers application to decouple the auditing and shipping logic from the checkout process. The `CheckoutController` class that you built invokes the `Auditing` and `Shipping` components by using a delegate and has no knowledge about these components or the methods it's running; this is the responsibility of the application that creates the

CheckoutController object and adds the appropriate references to the delegate. However, it might be useful for a component to be able to alert the application when it has completed its processing and enable the application to perform any necessary tidying up.

This might sound a little strange at first. Surely, when the application invokes the delegate in the CheckoutController object, the methods referenced by this delegate run, and the application continues with the next statement only when these methods have finished. But this is not necessarily the case! Chapter 24, “Improving response time by performing asynchronous operations,” demonstrates that methods can run asynchronously, and when you invoke a method, it might not have completed before execution continues with the next statement. This is especially true in UWP apps in which long-running operations are performed on background threads to enable the user interface to remain responsive.

In the Wide World Importers application, in the CheckoutButtonClicked method, the code that invokes the delegate is followed by a statement that displays a dialog with a message indicating that the order has been placed:

```
private void CheckoutButtonClicked(object sender, RoutedEventArgs e)
{
    try
    {
        // Perform the checkout processing
        this.checkoutController.StartCheckoutProcessing(this.order);

        // Display a summary of the order
        MessageDialog dlg = new MessageDialog(...);
        _ = dlg.ShowAsync();
        ...
    }
    ...
}
```

In fact, there’s no guarantee that the processing performed by the delegated methods has completed by the time the dialog appears, so the message could actually be misleading. This is where an event is invaluable. The Auditor and Shipper components could both publish an event to which the application subscribes. This event could be raised by the components only when they have completed their processing. When the application receives this event, it can display the message, safe in the knowledge that it’s now accurate.

In the following exercise, you’ll modify the Auditor and Shipper classes to raise an event that occurs when they’ve completed their processing. The application will subscribe to the event for each component and display an appropriate message when the event occurs.

To add an event to the CheckoutController class

1. Return to Visual Studio 2022 and display the Delegates solution.
2. In the AuditService project, open the Auditor.cs file in the Code and Text Editor window.

3. Add a public delegate called `AuditingCompleteDelegate` to the `Auditor` class. This delegate should specify a method that takes a string parameter called `message` and that returns a void. The code in bold in the following example shows the definition of this delegate:

```
public class Auditor
{
    public delegate void AuditingCompleteDelegate(string message);
    ...
}
```

4. Add a public event called `AuditProcessingComplete` to the `Auditor` class, after the `AuditingCompleteDelegate` delegate. This event should be based on the `AuditingCompleteDelegate` delegate as shown in bold in the following code:

```
public class Auditor
{
    public delegate void AuditingCompleteDelegate(string message);
    public event AuditingCompleteDelegate AuditProcessingComplete;
    ...
}
```

5. Locate the `AuditOrder` method. This is the method that is run by using the delegate in the `CheckoutController` object. It invokes another private method called `doAuditing` to actually perform the audit operation. The method looks like this:

```
public void AuditOrder(Order order)
{
    this.doAuditing(order);
}
```

6. Scroll down to the `doAuditing` method. The code in this method is enclosed in a `try/catch` block; it uses the XML APIs of the .NET class library to generate an XML representation of the order being audited and saves it to a file. (The exact details of how this works are beyond the scope of this book.)
7. After the catch block, add a `finally` block that raises the `AuditProcessingComplete` event, as shown in bold in the following:

```
private async void doAuditing(Order order)
{
    List<OrderItem> ageRestrictedItems = findAgeRestrictedItems(order);
    if (ageRestrictedItems.Count > 0)
    {
        try
        {
            ...
        }
        catch (Exception ex)
        {
            ...
        }
        finally
        {
```

```

        if (this.AuditProcessingComplete is not null)
        {
            this.AuditProcessingComplete($"Audit record written for Order
{order.OrderID}");
        }
    }
}

```

8. In the DeliveryService project, open the Shipper.cs file in the Code and Text Editor window.
9. Add a public delegate called `ShippingCompleteDelegate` to the `Shipper` class. This delegate should specify a method that takes a string parameter called `message` and that returns a `void`. The code in bold in the following example shows the definition of this delegate:

```

public class Shipper
{
    public delegate void ShippingCompleteDelegate(string message);
    ...
}

```

10. Add a public event called `ShipProcessingComplete` to the `Shipper` class, based on the `ShippingCompleteDelegate` delegate as shown in bold in the following code:

```

public class Shipper
{
    public delegate void ShippingCompleteDelegate(string message);
    public event ShippingCompleteDelegate ShipProcessingComplete;
    ...
}

```

11. Find the `doShipping` method, which is the method that performs the shipping logic. In the method, after the catch block, add a `finally` block that raises the `ShipProcessingComplete` event, as shown here in bold:

```

private async void doShipping(Order order)
{
    try
    {
        ...
    }
    catch (Exception ex)
    {
        ...
    }
    finally
    {
        if (this.ShipProcessingComplete is not null)
        {
            this.ShipProcessingComplete($"Dispatch note generated for Order
{order.OrderID}");
        }
    }
}

```


12. In the Delegates project, display the layout for the MainPage.xaml file in the Design View window. Then, in the XAML pane, scroll down to the first set of RowDefinition items. The XAML code looks like this:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <Grid Margin="12,0,12,0" Loaded="MainPageLoaded">
    <Grid.RowDefinitions>
      <RowDefinition Height="*" />
      <RowDefinition Height="2*" />
      <RowDefinition Height="*" />
      <RowDefinition Height="10*" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    ...
  </Grid>
</Grid>
```

13. Change the Height property of the final RowDefinition item to 2* as shown in bold in the following code:

```
<Grid.RowDefinitions>
  ...
  <RowDefinition Height="10*" />
  <RowDefinition Height="2*" />
</Grid.RowDefinitions>
```

This change in the layout makes a bit of space available at the bottom of the form. You'll use this space to display the messages received from the Auditor and Shipper components when they raise their events. Chapter 25, "Implementing the user interface for a Universal Windows Platform app," provides more detail on laying out user interfaces by using a Grid control.

14. Scroll to the bottom of the XAML pane and add the following ScrollView and TextBlock elements shown in bold before the penultimate </Grid> tag:

```
...
</Grid>
<ScrollView Grid.Row="4" VerticalScrollBarVisibility="Visible">
  <TextBlock x:Name="messageBar" FontSize="18" />
</ScrollView>
</Grid>
</Grid>
</Page>
```

This markup adds a TextBlock control called messageBar to the area at the bottom of the screen. You'll use this control to display messages from the Auditor and Shipper objects. Again, you'll learn more about grid layouts in Chapter 25.

15. Display the MainPage.xaml.cs file in the Code and Text Editor window. Then find the CheckoutButtonClicked method and remove the code that displays the summary of the order. The try block should look like this after you've deleted the code:

```
private void CheckoutButtonClicked(object sender, RoutedEventArgs e)
{
    try
    {
        // Perform the checkout processing
        this.checkoutController.StartCheckoutProcessing(this.order);
    }
}
```

```

        // Deleted code was here

        // Clear out the order details so the user can start again with a new order
        this.order = new Order
        {
            Date = DateTime.Now,
            Items = new List<OrderItem>(),
            OrderID = Guid.NewGuid(),
            TotalValue = 0
        };
        this.orderDetails.DataContext = null;
        this.orderValue.Text = $"{order.TotalValue:C}";
        this.listViewHeader.Visibility = Visibility.Collapsed;
        this.checkout.IsEnabled = false;
    }
    catch (Exception ex)
    {
        ...
    }
}

```

- 16.** Add a private method called `displayMessage` to the `MainPage` class. This method should take a string parameter called `message` and should return a `void`. In the body of this method, add a statement that appends the value in the `message` parameter to the `Text` property of the `messageBar TextBlock` control, followed by a newline character, as shown here in bold:

```

private void displayMessage(string message)
{
    this.messageBar.Text += $"{message}{Environment.NewLine}";
}

```

This code causes the message to appear in the message area at the bottom of the form.

- 17.** Find the constructor for the `MainPage` class and add the code shown here in bold:

```

public MainPage()
{
    ...
    this.auditor = new Auditor();
    this.shipper = new Shipper();
    this.checkoutController = new CheckoutController();
    this.checkoutController.CheckoutProcessing += this.auditor.AuditOrder;
    this.checkoutController.CheckoutProcessing += this.shipper.ShipOrder;
    this.auditor.AuditProcessingComplete += this.displayMessage;
    this.shipper.ShipProcessingComplete += this.displayMessage;
}

```

These statements subscribe to the events exposed by the `Auditor` and `Shipper` objects. When the events are raised, the `displayMessage` method runs. Notice that the same method handles both events.

- 18.** On the **Debug** menu, select **Start Debugging** to build and run the application.
- 19.** When the Wide World Importers form appears, select some items (include at least one age-restricted item), and then select **Checkout**.

20. Verify that the “Audit record written” message appears in the area at the bottom of the form, followed by the “Dispatch note generated” message.

Delegates

Wide World Importers

Hammer	Name: Power Saw	Order Details
24oz heavy-duty claw hammer	Description: Rotary action, high powered	
£18.35	Price: £88.00	
No	Age Restricted: Yes	
	Add	
Audit record written for Order 24975f99-cddf-4521-850d-2547ccde3dda Dispatch note generated for Order 24975f99-cddf-4521-850d-2547ccde3dda		Total: £0.00 Checkout

21. Place further orders and note the new messages that appear each time you select Checkout. You might need to scroll down to see them when the message area fills up.
22. When you’ve finished, return to Visual Studio 2022 and stop debugging.

Summary

In this chapter, you learned how to use delegates to reference methods and invoke those methods. You also saw how to define lambda expressions that can be run by using a delegate. Finally, you learned how to define and use events to trigger execution of a method.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 21.
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

To	Do this
Declare a delegate type	Write the keyword <code>delegate</code> , followed by the return type, followed by the name of the delegate type, followed by any parameter types. For example: <code>delegate void myDelegate();</code>
Create an instance of a delegate initialized with a single specific method	Use the same syntax you use for a class or structure: write the keyword <code>new</code> , followed by the name of the type (the name of the delegate), followed by the argument between parentheses. The argument must be a method whose signature exactly matches the signature of the delegate. For example: <code>delegate void myDelegate();</code> <code>private void myMethod() { ... }</code> <code>...</code> <code>myDelegate del = new myDelegate(this.myMethod);</code>
Invoke a delegate	Use the same syntax as a method call. For example: <code>myDelegate del;</code> <code>...</code> <code>del();</code>
Declare an event	Write the keyword <code>event</code> , followed by the name of the type (the type must be a delegate type), followed by the name of the event. For example: <code>class MyClass</code> <code>{</code> <code> public delegate void MyDelegate();</code> <code> ...</code> <code> public event myDelegate MyEvent;</code> <code>}</code>
Subscribe to an event	Create a delegate instance (of the same type as the event) and attach the delegate instance to the event by using the <code>+=</code> operator. For example: <code>class MyEventHandlerClass</code> <code>{</code> <code> private MyClass myClass = new MyClass();</code> <code> ...</code> <code> public void Start()</code> <code> {</code> <code> myClass.MyEvent +=</code> <code> new myClass.MyDelegate (this.eventHandlingMethod);</code> <code> }</code> <code> private void eventHandlingMethod()</code> <code> {</code> <code> ...</code> <code> }</code> <code>}</code> You can also get the compiler to generate the new delegate automatically simply by specifying the subscribing method: <code>public void Start()</code> <code>{</code> <code> myClass.MyEvent +=</code> <code> this.eventHandlingMethod;</code> <code>}</code>

To	Do this
Unsubscribe from an event	Create a delegate instance (of the same type as the event), and detach it from the event by using the – operator. For example: <pre> class MyEventHandlerClass { private MyClass myClass = new MyClass(); ... public void Stop() { myClass.MyEvent -= new myClass.MyDelegate (this.eventHandlingMethod); } ... } </pre> Or: <pre> public void Stop() { myClass.MyEvent -= this.eventHandlingMethod; } </pre>
Raise an event	Use the same syntax as a method call. You must supply arguments to match the type of the parameters expected by the delegate referenced by the event. Don't forget to check whether the event is null. For example: <pre> class MyClass { public event myDelegate MyEvent; ... private void RaiseEvent() { if (this.MyEvent is not null) { this.MyEvent(); } } ... } </pre>

Querying in-memory data by using query expressions

After completing this chapter, you will be able to:

- Define Language-Integrated Query (LINQ) queries to examine the contents of enumerable collections.
- Use LINQ extension methods and query operators.
- Explain how LINQ defers evaluation of a query and how you can force immediate execution and cache the results of a LINQ query.

You've now met most of the features of the C# language. However, so far I have glossed over one important aspect of the language that will likely be used by many applications: the support that C# provides for querying data.

You've seen that you can define structures and classes to model data and that you can use collections and arrays to temporarily store data in memory. However, how do you perform common tasks such as searching for items in a collection that match a specific set of criteria? For example, if you have a collection of `Customer` objects, how do you find all customers that are located in New York, or which town has the most customers who have procured your services? You can write your own code to iterate through a collection and examine the fields in each object, but these types of tasks occur so often that the designers of C# decided to include features in the language to minimize the amount of code you need to write. In this chapter, you'll learn how to use these advanced C# language features to query and manipulate data.

What is LINQ?

All but the most trivial of applications need to process data. Historically, most applications provided their own logic for performing these operations. However, this strategy can lead to the code in an application becoming very tightly coupled with the structure of the data that it processes. If the data structures change, you might need to make a significant number of changes to the code that handles the data.

The designers of the Microsoft .NET Framework thought long and hard about this issue. They ultimately decided to make the life of an application developer easier by providing features that abstract the mechanism that applications use to query data from application code itself. These features are collectively called Language-Integrated Query, or LINQ.

The creators of LINQ took an unabashed look at how relational database management systems such as Microsoft SQL Server separate the language used to query a database from the internal format of the data in the database. Developers accessing a SQL Server database issue Structured Query Language (SQL) statements to the database management system. SQL provides a high-level description of the data that the developer wants to retrieve but does not indicate exactly how the database management system should retrieve it. These details are controlled by the database management system itself. Consequently, an application that invokes SQL statements does not care how the database management system physically stores or retrieves data. The format used by the database management system can change (for example, if a new version is released) without the application developer needing to modify the SQL statements used by the application.

LINQ provides syntax and semantics very reminiscent of SQL and with many of the same advantages. You can change the underlying structure of the data being queried without needing to change the code that actually performs the queries. But although LINQ looks similar to SQL, it is far more flexible and can handle a wider variety of logical data structures. For example, LINQ can handle data organized hierarchically, such as that found in an XML document. However, this chapter concentrates on using LINQ in a relational manner.

Using LINQ in a C# application

Perhaps the easiest way to explain how to use the C# features that support LINQ is to work through some simple examples based on the following sets of customer and address information:

Customer Information

CustomerID	FirstName	LastName	CompanyName
1	Kim	Abercrombie	Alpine Ski House
2	Jeff	Hay	Coho Winery
3	Charlie	Herb	Alpine Ski House
4	Chris	Preston	Trey Research
5	Dave	Barnett	Wingtip Toys
6	Ann	Beebe	Coho Winery
7	John	Kane	Wingtip Toys
8	David	Simpson	Trey Research
9	Greg	Chapman	Wingtip Toys
10	Tim	Litton	Wide World Importers

Address Information

CompanyName	City	Country
Alpine Ski House	Berne	Switzerland
Coho Winery	San Francisco	United States
Trey Research	New York	United States
Wingtip Toys	London	United Kingdom
Wide World Importers	Tetbury	United Kingdom

LINQ requires data to be stored in a data structure that implements the `IEnumerable` or `IEnumerable<T>` interface, as described in Chapter 19, “Enumerating collections.” It doesn’t matter what structure you use (an array, a `HashSet<T>`, a `Queue<T>`, or any other collection type, or even one that you define yourself) as long as it is enumerable. However, to keep things straightforward, the examples in this chapter assume that the customer and address information is held in the `customers` and `addresses` arrays shown in the following code example.



Note In a real-world application, you would populate these arrays by reading the data from a file or a database.

```
var customers = new[] {  
    new { CustomerID = 1, FirstName = "Kim", LastName = "Abercrombie",  
          CompanyName = "Alpine Ski House" },  
    new { CustomerID = 2, FirstName = "Jeff", LastName = "Hay",  
          CompanyName = "Coho Winery" },  
    new { CustomerID = 3, FirstName = "Charlie", LastName = "Herb",  
          CompanyName = "Alpine Ski House" },  
    new { CustomerID = 4, FirstName = "Chris", LastName = "Preston",  
          CompanyName = "Trey Research" },  
    new { CustomerID = 5, FirstName = "Dave", LastName = "Barnett",  
          CompanyName = "Wingtip Toys" },  
    new { CustomerID = 6, FirstName = "Ann", LastName = "Beebe",  
          CompanyName = "Coho Winery" },  
    new { CustomerID = 7, FirstName = "John", LastName = "Kane",  
          CompanyName = "Wingtip Toys" },  
    new { CustomerID = 8, FirstName = "David", LastName = "Simpson",  
          CompanyName = "Trey Research" },  
    new { CustomerID = 9, FirstName = "Greg", LastName = "Chapman",  
          CompanyName = "Wingtip Toys" },  
    new { CustomerID = 10, FirstName = "Tim", LastName = "Litton",  
          CompanyName = "Wide World Importers" }  
};  
  
var addresses = new[] {  
    new { CompanyName = "Alpine Ski House", City = "Berne",  
          Country = "Switzerland"},  
    new { CompanyName = "Coho Winery", City = "San Francisco",  
          Country = "United States"},  
    new { CompanyName = "Trey Research", City = "New York",  
          Country = "United States"},  
    new { CompanyName = "Wingtip Toys", City = "London",  
          Country = "United Kingdom"},  
};
```



```
new { CompanyName = "Wide World Importers", City = "Tetbury",  
      Country = "United Kingdom" }  
};
```



Note The sections “Selecting data,” “Filtering data,” “Ordering, grouping, and aggregating data,” and “Joining data” that follow show you the basic capabilities and syntax for querying data by using LINQ methods. The syntax can become a little complex at times, but you will see when you reach the section “Using query operators” that you don’t actually need to remember how it works. However, it’s useful for you to at least take a look at these sections so that you can fully appreciate how the query operators provided with C# perform their tasks.

Selecting data

Suppose that you want to display a list consisting of the first name of each customer in the `customers` array. You can achieve this task with the following code:

```
IEnumerable<string> customerFirstNames =  
    customers.Select(cust => cust.FirstName);  
foreach (string name in customerFirstNames)  
{  
    Console.WriteLine(name);  
}
```



Note The code for the examples shown in this section is available in the `LINQSamples` solution, located in the `\Microsoft Press\VC SBS\Chapter 21\LINQSamples` folder in your Documents folder.

Although this block of code is quite short, it does a lot, and it requires a degree of explanation—starting with the use of the `Select` method of the `customers` array.

Using the `Select` method, you can retrieve specific data from the array—in this case, the value in the `FirstName` field of each item in the array. How does it work? The parameter to the `Select` method is actually another method that takes a row from the `customers` array and returns the selected data from that row.

You can define your own custom method to perform this task, but the simplest mechanism is to use a lambda expression to define an anonymous method, as shown in the preceding example. There are three important things that you must understand at this point:

- The variable `cust` is the parameter passed into the method. You can think of `cust` as an alias for each row in the `customers` array. The compiler deduces this from the fact that you’re calling the `Select` method on the `customers` array. You can use any legal C# identifier in place of `cust`.
- The `Select` method doesn’t actually retrieve the data at this time; it simply returns an enumerable object that will fetch the data identified by the `Select` method when you iterate over it later. We’ll return to this aspect of LINQ in the section “LINQ and deferred evaluation” later in this chapter.

- The `Select` method is not actually a method of the `Array` type; it's an extension method of the `Enumerable` class. The `Enumerable` class is located in the `System.Linq` namespace and provides a substantial set of static methods for querying objects that implement the generic `IEnumerable<T>` interface.

The preceding example uses the `Select` method of the `customers` array to generate an `IEnumerable<string>` object named `customerFirstNames`. (It is of type `IEnumerable<string>` because the `Select` method returns an enumerable collection of customer first names, which are strings.) The `foreach` statement iterates through this collection of strings, printing out the first name of each customer in the following sequence:

```
Kim
Jeff
Charlie
Chris
Dave
Ann
John
David
Greg
Tim
```

You can now display the first name of each customer. Fetching the first and last names of each customer is slightly trickier, however. If you examine the definition of the `Enumerable.Select` method in the `System.Linq` namespace in the documentation supplied with Microsoft Visual Studio 2022, you'll see that it looks like this:

```
public static IEnumerable<TResult> Select<TSource, TResult> (
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector
)
```

What this actually says is that `Select` is a generic method that takes two type parameters named `TSource` and `TResult` as well as two ordinary parameters named `source` and `selector`. `TSource` is the type of the collection for which you're generating an enumerable set of results (customer objects in this example), and `TResult` is the type of the data in the enumerable set of results (string objects in this example). Remember that `Select` is an extension method, so the `source` parameter is actually a reference to the type being extended (a generic collection of customer objects that implements the `IEnumerable` interface in the example). The `selector` parameter specifies a generic method that identifies the fields to be retrieved. (Recall that `Func` is the name of a generic delegate type in the .NET Framework that you can use for encapsulating a generic method that returns a result.) The method referred to by the `selector` parameter takes a `TSource` (in this case, `customer`) parameter and yields a `TResult` (in this case, `string`) object. The value returned by the `Select` method is an enumerable collection of `TResult` (again `string`) objects.



Note Chapter 12, "Working with inheritance," explains how extension methods work and the role of the first parameter to an extension method.

The important point to understand is that the `Select` method returns an enumerable collection based on a single type. If you want the enumerator to return multiple items of data, such as the first and last names of each customer, you have at least two options:

- You can concatenate the first and last names together into a single string in the `Select` method, like this:

```
IEnumerable<string> customerNames =  
    customers.Select(cust => $"{cust.FirstName} {cust.LastName}");
```

- You can define a new type that wraps the first and last names and use the `Select` method to construct instances of this type, like this:

```
class FullName  
{  
    public string FirstName{ get; set; }  
    public string LastName{ get; set; }  
}  
...  
IEnumerable<FullName> customerFullNames =  
    customers.Select(cust => new FullName  
    {  
        FirstName = cust.FirstName,  
        LastName = cust.LastName  
    });
```

The second option is arguably preferable, but if this is the only use that your application makes of the `Names` type, you might prefer to use an anonymous type, as in the following code, instead of defining a new type specifically for a single operation:

```
var customerFullNames =  
    customers.Select(cust => new  
{  
        FirstName = cust.FirstName,  
        LastName = cust.LastName  
});
```

Notice the use of the `var` keyword here to define the type of the enumerable collection. The type of objects in the collection is anonymous, so you don't know the specific type of the objects in the collection.

Filtering data

With the `Select` method, you can specify, or *project*, the fields that you want to include in the enumerable collection. However, you might also want to restrict the rows that the enumerable collection contains. For example, suppose you want to list only the names of companies in the `addresses` array that are located in the United States. To do this, you can use the `Where` method, as follows:

```
IEnumerable<string> usCompanies = addresses  
    .Where(addr => String.Equals(addr.Country, "United States"))  
    .Select(usComp => usComp.CompanyName);
```

```
foreach (string name in usCompanies)
{
    Console.WriteLine(name);
}
```

Syntactically, the `Where` method is similar to `Select`. It expects a parameter that defines a method that filters the data according to whatever criteria you specify. This example uses another lambda expression. The variable `addr` is an alias for a row in the `addresses` array, and the lambda expression returns all rows where the `Country` field matches the string `United States`. The `Where` method returns an enumerable collection of rows containing every field from the original collection. The `Select` method is then applied to these rows to project only the `CompanyName` field from this enumerable collection to return another enumerable collection of `string` objects. (The variable `usComp` is an alias for the type of each row in the enumerable collection returned by the `Where` method.) The type of the result of this complete expression is therefore `IEnumerable<string>`. It's important to understand this sequence of operations—the `Where` method is applied first to filter the rows, followed by the `Select` method to specify the fields. The `foreach` statement that iterates through this collection displays the following companies:

```
Coho Winery
Trey Research
```

Ordering, grouping, and aggregating data

If you're familiar with SQL, you know that it makes it possible for you to perform a wide variety of relational operations beyond simple projection and filtering. For example, you can specify that you want data to be returned in a specific order, group the rows returned according to one or more key fields, and calculate summary values based on the rows in each group. LINQ provides the same functionality.

To retrieve data in a particular order, you can use the `OrderBy` method. Like the `Select` and `Where` methods, `OrderBy` expects a method as its argument. This method identifies the expressions that you want to use to sort the data. For example, you can display the name of each company in the `addresses` array in ascending order, like this:

```
IEnumerable<string> companyNames = addresses
    .OrderBy(addr => addr.CompanyName)
    .Select(comp => comp.CompanyName);

foreach (string name in companyNames)
{
    Console.WriteLine(name);
}
```

This block of code displays the companies in the `addresses` table in alphabetical order.

```
Alpine Ski House
Coho Winery
Trey Research
Wide World Importers
Wingtip Toys
```

If you want to enumerate the data in descending order, you can use the `OrderByDescending` method instead. If you want to order by more than one key value, you can use the `ThenBy` or `ThenByDescending` method after `OrderBy` or `OrderByDescending`.

To group data according to common values in one or more fields, you can use the `GroupBy` method. The following example shows how to group the companies in the `addresses` array by country:

```
var companiesGroupedByCountry = addresses
    .GroupBy(addr => addr.Country);

foreach (var companiesPerCountry in companiesGroupedByCountry)
{
    Console.WriteLine(
        $"Country: {companiesPerCountry.Key}\t{companiesPerCountry.Count()} companies");

    foreach (var companies in companiesPerCountry)
    {
        Console.WriteLine($"    \t{companies.CompanyName}");
    }
}
```

By now, you should recognize the pattern. The `GroupBy` method expects a method that specifies the fields by which to group the data. However, there are some subtle differences between the `GroupBy` method and the other methods that you've seen so far.

The main point of interest is that you don't need to use the `Select` method to project the fields to the result. The enumerable set returned by `GroupBy` contains all the fields in the original source collection, but the rows are ordered into a set of enumerable collections based on the field identified by the method specified by `GroupBy`. In other words, the result of the `GroupBy` method is an enumerable set of groups, each of which is an enumerable set of rows.

In the example just shown, the enumerable set `companiesGroupedByCountry` is a set of countries. The items in this set are themselves enumerable collections containing the companies for each country in turn. The code that displays the companies in each country uses a `foreach` loop to iterate through the `companiesGroupedByCountry` set to yield and display each country in turn, and then it uses a nested `foreach` loop to iterate through the set of companies in each country.

Notice in the outer `foreach` loop that you can access the value that you're grouping by using the `Key` field of each item. You can also calculate summary data for each group by using methods such as `Count`, `Max`, `Min`, and many others. The output generated by the code example looks like this:

```
Country: Switzerland    1 companies
        Alpine Ski House
Country: United States  2 companies
        Coho Winery
        Trey Research
Country: United Kingdom 2 companies
        Wingtip Toys
        Wide World Importers
```

You can use many of the summary methods such as `Count`, `Max`, and `Min` directly over the results of the `Select` method. For example, if you want to know how many companies there are in the `addresses` array, you can use a block of code such as this:

```
int numberOfCompanies = addresses
    .Select(addr => addr.CompanyName).Count();
Console.WriteLine($"Number of companies: {numberOfCompanies}");
```

Notice that the result of these methods is a single scalar value rather than an enumerable collection. The output from the preceding block of code looks like this:

```
Number of companies: 5
```

I should utter a word of caution at this point. These summary methods do not distinguish between rows in the underlying set that contain duplicate values in the fields you are projecting. This means that strictly speaking, the preceding example shows you only how many rows in the `addresses` array contain a value in the `CompanyName` field. If you wanted to find out how many different countries are mentioned in this table, you might be tempted to try this:

```
int numberOfCountries = addresses
    .Select(addr => addr.Country).Count();
Console.WriteLine($"Number of countries: {numberOfCountries}");
```

The output looks like this:

```
Number of countries: 5
```

In fact, the `addresses` array includes only three different countries; it just so happens that United States and United Kingdom both occur twice. You can eliminate duplicates from the calculation by using the `Distinct` method, like this:

```
int numberOfDistinctCountries = addresses
    .Select(addr => addr.Country).Distinct().Count();
Console.WriteLine($"Number of distinct countries: {numberOfDistinctCountries}");
```

The `Console.WriteLine` statement now outputs the expected result:

```
Number of countries: 3
```

Joining data

Just like SQL, LINQ gives you the ability to join multiple sets of data over one or more common key fields. The following example shows how to display the first and last names of each customer, with the name of the country where the customer is located:

```
var companiesAndCustomers = customers
    .Select(c => new { c.FirstName, c.LastName, c.CompanyName })
    .Join(addresses, custs => custs.CompanyName, addr => addr.CompanyName,
        (custs, addr) => new {custs.FirstName, custs.LastName, addr.Country });

foreach (var row in companiesAndCustomers)
{
    Console.WriteLine(row);
}
```

The customers' first and last names are available in the `customers` array, but the country for each company that customers work for is stored in the `addresses` array. The common key between the `customers` array and the `addresses` array is the company name (`CompanyName`). The `Select` method specifies the fields of interest in the `customers` array (`FirstName` and `LastName`) and the field containing the common key (`CompanyName`).

You use the `Join` method to join the data identified by the `Select` method with another enumerable collection. The parameters to the `Join` method are as follows:

- The enumerable collection to join with
- A method that identifies the common key fields from the data identified by the `Select` method
- A method that identifies the common key fields to join the selected data on
- A method that specifies the columns you require in the enumerable result set returned by the `Join` method

In this example, the `Join` method joins the enumerable collection containing the `FirstName`, `LastName`, and `CompanyName` fields from the `customers` array with the rows in the `addresses` array. The two sets of data are joined where the value in the `CompanyName` field in the `customers` array matches the value in the `CompanyName` field in the `addresses` array. The result set includes rows containing the `FirstName` and `LastName` fields from the `customers` array with the `Country` field from the `addresses` array. The code that outputs the data from the `companiesAndCustomers` collection displays the following information:

```
{ FirstName = Kim, LastName = Abercrombie, Country = Switzerland }  
{ FirstName = Jeff, LastName = Hay, Country = United States }  
{ FirstName = Charlie, LastName = Herb, Country = Switzerland }  
{ FirstName = Chris, LastName = Preston, Country = United States }  
{ FirstName = Dave, LastName = Barnett, Country = United Kingdom }  
{ FirstName = Ann, LastName = Beebe, Country = United States }  
{ FirstName = John, LastName = Kane, Country = United Kingdom }  
{ FirstName = David, LastName = Simpson, Country = United States }  
{ FirstName = Greg, LastName = Chapman, Country = United Kingdom }  
{ FirstName = Tim, LastName = Litton, Country = United Kingdom }
```



Note Collections in memory are not the same as tables in a relational database, and the data they contain is not subject to the same data-integrity constraints. In a relational database, it could be acceptable to assume that every customer has a corresponding company and that each company has its own unique address. Collections don't enforce the same level of data integrity, meaning that you can quite easily have a customer referencing a company that doesn't exist in the `addresses` array, and you might even have the same company occurring more than once in the `addresses` array. In these situations, the results that you obtain might be accurate but unexpected. Join operations work best when you fully understand the relationships between the data you are joining.

Using query operators

The preceding sections have shown you many of the features available for querying in-memory data by using the extension methods for the `Enumerable` class defined in the `System.Linq` namespace. The syntax uses several advanced C# language features, and the resultant code can sometimes be quite hard

to understand and maintain. To relieve you of some of this burden, the designers of C# added query operators to the language with which you can employ LINQ features by using a syntax more akin to SQL.

As you saw in the examples earlier in this chapter, you can retrieve the first name for each customer like this:

```
IEnumerable<string> customerFirstNames = customers
    .Select(cust => cust.FirstName);
```

You can rephrase this statement by using the `from` and `select` query operators, like this:

```
var customerFirstNames = from cust in customers
    select cust.FirstName;
```

At compile time, the C# compiler resolves this expression into the corresponding `Select` method. The `from` operator defines an alias for the source collection, and the `select` operator specifies the fields to retrieve using this alias. The result is an enumerable collection of customer first names. If you're familiar with SQL, notice that the `from` operator occurs before the `select` operator.

Continuing in the same vein, to retrieve the first and last names for each customer, you can use the following statement. (You might want to refer to the earlier example of the same statement based on the `Select` extension method.)

```
var customerNames = from cust in customers
    select new { cust.FirstName, cust.LastName };
```

You use the `where` operator to filter data. The following example shows how to return the names of the companies based in the United States from the `addresses` array:

```
var usCompanies = from a in addresses
    where String.Equals(a.Country, "United States")
    select a.CompanyName;
```

To order data, you use the `orderby` operator, like this:

```
var companyNames = from a in addresses
    orderby a.CompanyName
    select a.CompanyName;
```

You can group data by using the `group` operator in the following manner:

```
var companiesGroupedByCountry = from a in addresses
    group a by a.Country;
```

Notice that, as with the earlier example showing how to group data, you do not provide the `select` operator. You can iterate through the results by using the same code as the earlier example, like this:

```
foreach (var companiesPerCountry in companiesGroupedByCountry)
{
    Console.WriteLine(
        $"Country: {companiesPerCountry.Key}\t{companiesPerCountry.Count()} companies");
    foreach (var companies in companiesPerCountry)
    {
        Console.WriteLine($" \t{companies.CompanyName}");
    }
}
```


You can invoke summary functions such as `Count` over the collection returned by an enumerable collection like this:

```
int numberOfCompanies = (from a in addresses
                        select a.CompanyName).Count();
```

Notice that you wrap the expression in parentheses. If you want to ignore duplicate values, use the `Distinct` method:

```
int numberOfCountries = (from a in addresses
                        select a.Country).Distinct().Count();
```



Tip In many cases, you probably want to count just the number of rows in a collection rather than the number of values in a field across all the rows in the collection. In this case, you can invoke the `Count` method directly over the original collection, like this:

```
int numberOfCompanies = addresses.Count();
```

You can use the `join` operator to combine two collections across a common key. The following example shows the query returning customers and addresses over the `CompanyName` column in each collection, this time rephrased by using the `join` operator. You use the `on` clause with the `equals` operator to specify how the two collections are related.



Note LINQ currently supports equi-joins (joins based on equality) only. If you're a database developer accustomed to SQL, you might be familiar with joins based on other operators, such as `>` and `<`, but LINQ does not provide these features.

```
var countriesAndCustomers = from a in addresses
                          join c in customers
                          on a.CompanyName equals c.CompanyName
                          select new { c.FirstName, c.LastName, a.Country };
```



Note In contrast with SQL, the order of the expressions in the `on` clause of a LINQ expression is important. You must place the item you're joining from (referencing the data in the collection in the `from` clause) to the left of the `equals` operator and the item you are joining with (referencing the data in the collection in the `join` clause) to the right.

LINQ provides numerous other methods for summarizing information and joining, grouping, and searching through data. This section has covered just the most common features. For example, LINQ provides the `Intersect` and `Union` methods, which you can use to perform set-wide operations. It also provides methods such as `Any` and `All` that you can use to determine whether at least one item in a collection or every item in a collection matches a specified predicate. You can partition the values in an enumerable collection by using the `Take` and `Skip` methods. For more information, see the material in the LINQ section of the documentation provided with Visual Studio 2022.

Querying data in Tree<TItem> objects

The examples you've seen so far in this chapter have shown how to query the data in an array. You can use the same techniques for any collection class that implements the generic `IEnumerable<T>` interface. In the following exercise, you'll define a new class for modeling employees for a company. You'll create a `BinaryTree` object containing a collection of `Employee` objects, and then you'll use LINQ to query this information. You will initially call the LINQ extension methods directly, but then you'll modify your code to use query operators.

To retrieve data from a `BinaryTree` by using the extension methods

1. Start Visual Studio 2022, if it is not already running.
2. Open the **QueryBinaryTree** solution, which is located in the `\Microsoft Press\VCSBS\Chapter 21\QueryBinaryTree` folder in your **Documents** folder.

The project in the solution contains the `Program.cs` file, which defines the `Program` class with the `Main` and `doWork` methods that you've seen in previous exercises. The solution also includes a copy of the `BinaryTree` project that you've seen in previous chapters.

3. In Solution Explorer, right-click the **QueryBinaryTree** project, point to **Add**, and then select **Class**.
4. In the Add New Item – Query BinaryTree dialog, type **Employee.cs** in the **Name** box, and then select **Add**.
5. Add the automatic properties shown in bold in the following code to the `Employee` class:

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Department { get; set; }
    public int Id { get; set; }
}
```

6. Add the `ToString` method shown in bold in the code that follows to the `Employee` class. Types in the .NET Framework use this method when converting the object to a string representation, such as when displaying it by using the `Console.WriteLine` statement.

```
class Employee
{
    ...
    public override string ToString() =>
         $"Id: {this.Id}, Name: {this.FirstName} {this.LastName}, Dept: {this.Department}";
}
```

7. Modify the definition of the `Employee` class to implement the `IComparable<Employee>` interface, as shown here:

```
class Employee : IComparable<Employee>
{
}
```

This step is necessary because the `BinaryTree` class specifies that its elements must be “comparable.”

8. Hover over the `IComparable<Employee>` interface in the class definition, select the lightbulb icon that appears, and then select **Implement all members explicitly** on the context menu.

This generates a default implementation of the `CompareTo` method. Remember that the `BinaryTree` class calls this method when it needs to compare elements when inserting them into the tree.

9. Replace the body of the `CompareTo` method with the following code shown in bold. This implementation of the `CompareTo` method compares `Employee` objects based on the value of the `Id` field.

```
int IComparable<Employee>.CompareTo(Employee other)
{
    if (other is null)
    {
        return 1;
    }

    if (this.Id > other.Id)
    {
        return 1;
    }

    if (this.Id < other.Id)
    {
        return -1;
    }

    return 0;
}
```



Note For a description of the `IComparable<T>` interface, refer to Chapter 19.

10. Display the `Program.cs` file for the `QueryBinaryTree` project in the Code and Text Editor window. Then add the following `using` directives to the top of the file:

```
using System.Linq;
using BinaryTree;
```

11. In the `doWork` method in the `Program` class, remove the `// TODO: comment` and add the following statements shown in bold to construct and populate an instance of the `BinaryTree` class:

```
static void doWork()
{
    Tree<Employee> empTree = new Tree<Employee>(
        new Employee { Id = 1, FirstName = "Kim", LastName = "Abercrombie", Department = "IT"
    });

    empTree.Insert(
        new Employee { Id = 2, FirstName = "Jeff", LastName = "Hay", Department = "Marketing"
    });
}
```

```

empTree.Insert(
    new Employee { Id = 4, FirstName = "Charlie", LastName = "Herb", Department = "IT"
});

empTree.Insert(
    new Employee { Id = 6, FirstName = "Chris", LastName = "Preston", Department = "Sales"
});

empTree.Insert(
    new Employee { Id = 3, FirstName = "Dave", LastName = "Barnett", Department = "Sales"
});

empTree.Insert(
    new Employee { Id = 5, FirstName = "Tim", LastName = "Litton", Department = "Marketing"
});
}

```

- 12.** Add the following statements shown in bold to the end of the `doWork` method. This code invokes the `Select` method to list the departments found in the binary tree.

```

static void doWork()
{
    ...
    Console.WriteLine("List of departments");
    var depts = empTree.Select(d => d.Department);

    foreach (var dept in depts)
    {
        Console.WriteLine($"Department: {dept}");
    }
}

```

- 13.** On the **Debug** menu, select **Start Without Debugging**.

The application should output the following list of departments:

```

List of departments
Department: IT
Department: Marketing
Department: Sales
Department: IT
Department: Marketing
Department: Sales

```

Each department occurs twice because there are two employees in each department. The order of the departments is determined by the `CompareTo` method of the `Employee` class, which uses the `Id` property of each employee to sort the data. The first department is for the employee with the `Id` value 1, the second department is for the employee with the `Id` value 2, and so on.

- 14.** Press **Enter** to return to Visual Studio 2022.

- 15.** In the `doWork` method in the `Program` class, modify the statement that creates the enumerable collection of departments as shown in bold in the following example:

```

var depts = empTree.Select(d => d.Department).Distinct();

```

The `Distinct` method removes duplicate rows from the enumerable collection.

16. On the **Debug** menu, select **Start Without Debugging**. Then verify that the application now displays each department only once, like this:

```
List of departments
Department: IT
Department: Marketing
Department: Sales
```

17. Press **Enter** to return to Visual Studio 2022.

18. Add the following statements shown in bold to the end of the `doWork` method. This block of code uses the `Where` method to filter the employees and return only those in the IT department. The `Select` method returns the entire row rather than projecting specific columns.

```
static void doWork()
{
    ...
    Console.WriteLine();
    Console.WriteLine("Employees in the IT department");
    var ITEmployees =
        empTree.Where(e => String.Equals(e.Department, "IT"))
        .Select(emp => emp);

    foreach (var emp in ITEmployees)
    {
        Console.WriteLine(emp);
    }
}
```

19. After the code from the preceding step, add the following code shown in bold to the end of the `doWork` method. This code uses the `GroupBy` method to group the employees found in the binary tree by department. The outer `foreach` statement iterates through each group, displaying the name of the department. The inner `foreach` statement displays the names of the employees in each department.

```
static void doWork()
{
    ...
    Console.WriteLine("");
    Console.WriteLine("All employees grouped by department");
    var employeesByDept = empTree.GroupBy(e => e.Department);

    foreach (var dept in employeesByDept)
    {
        Console.WriteLine($"Department: {dept.Key}");
        foreach (var emp in dept)
        {
            Console.WriteLine($"{emp.FirstName} {emp.LastName}");
        }
    }
}
```

20. On the **Debug** menu, select **Start Without Debugging**. Verify that the output of the application looks like this:

```
List of departments
Department: IT
Department: Marketing
Department: Sales

Employees in the IT department
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 4, Name: Charlie Herb, Dept: IT

All employees grouped by department
Department: IT
    Kim Abercrombie
    Charlie Herb
Department: Marketing
    Jeff Hay
    Tim Litton
Department: Sales
    Dave Barnett
    Chris Preston
```

21. Press **Enter** to return to Visual Studio 2022.

To retrieve data from a `BinaryTree` by using query operators

1. In the `doWork` method, comment out the statement that generates the enumerable collection of departments and replace it with the equivalent statement shown in bold, using the `from` and `select` query operators:

```
// var depts = empTree.Select(d => d.Department).Distinct();
var depts = (from d in empTree
             select d.Department).Distinct();
```

2. Comment out the statement that generates the enumerable collection of employees in the IT department and replace it with the following code shown in bold:

```
// var ITEmployees =
//     empTree.Where(e => String.Equals(e.Department, "IT"))
//     .Select(emp => emp);
var ITEmployees = from e in empTree
                  where String.Equals(e.Department, "IT")
                  select e;
```

3. Comment out the statement that generates the enumerable collection that groups employees by department and replace it with the statement shown in bold in the following code:

```
// var employeesByDept = empTree.GroupBy(e => e.Department);
var employeesByDept = from e in empTree
                      group e by e.Department;
```

4. On the **Debug** menu, select **Start Without Debugging**. Then verify that the program displays the same results as before.

```
List of departments
Department: IT
Department: Marketing
Department: Sales
Employees in the IT department
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 4, Name: Charlie Herb, Dept: IT
All employees grouped by department
Department: IT
    Kim Abercrombie
    Charlie Herb
Department: Marketing
    Jeff Hay
    Tim Litton
Department: Sales
    Dave Barnett
    Chris Preston
```

5. Press **Enter** to return to Visual Studio 2022.

LINQ and deferred evaluation

When you use LINQ to define an enumerable collection, either by using the LINQ extension methods or by using query operators, remember that the application doesn't actually build the collection at the time that the LINQ extension method is executed; instead, the collection is enumerated only when you iterate over it. This means that the data in the original collection can change in the time between the execution of a LINQ query and when the data that the query identifies is retrieved. However, you will always fetch the most up-to-date data. For example, the following query (which you saw earlier) defines an enumerable collection of companies in the United States:

```
var usCompanies = from a in addresses
                   where String.Equals(a.Country, "United States")
                   select a.CompanyName;
```

The data in the `addresses` array is not retrieved, and any conditions specified in the `Where` filter are not evaluated, until you iterate through the `usCompanies` collection:

```
foreach (string name in usCompanies)
{
    Console.WriteLine(name);
}
```

If you modify the data in the `addresses` array in the time between defining the `usCompanies` collection and iterating through the collection (for example, if you add a new company based in the United States), you'll see this new data. This strategy is referred to as *deferred evaluation*.

You can force the evaluation of a LINQ query when it is defined and generate a static, cached collection. This collection is a copy of the original data and will not change if the data in the collection changes. LINQ provides the `ToList` method to build a static `List` object containing a cached copy of the data. You use it like this:

```
var usCompanies = from a in addresses.ToList()
                  where String.Equals(a.Country, "United States")
                  select a.CompanyName;
```

This time, the list of companies is fixed when you create the query. If you add more United States companies to the `addresses` array, you won't see them when you iterate through the `usCompanies` collection. LINQ also provides the `ToArray` method, which stores the cached collection as an array.

In the final exercise in this chapter, you'll compare the effects of using deferred evaluation of a LINQ query to generating a cached collection.

To examine the effects of deferred and cached evaluation of a LINQ query

1. Return to Visual Studio 2022, display the `QueryBinaryTree` project, and then edit the `Program.cs` file.
2. Comment out the contents of the `doWork` method apart from the statements that construct the `empTree` binary tree, as shown here:

```
static void doWork()
{
    Tree<Employee> empTree = new Tree<Employee>(
        new Employee { Id = 1, FirstName = "Kim", LastName = "Abercrombie", Department = "IT"
    });

    ...
    empTree.Insert(
        new Employee { Id = 5, FirstName = "Tim", LastName = "Litton", Department = "Marketing"
    });

    /* comment out the rest of the method
        ...
        */
}
```



Tip You can comment out a block of code by selecting the entire block in the Code and Text Editor window and then selecting the `Comment Out The Selected Lines` button on the toolbar.

3. Add the following statements shown in bold to the `doWork` method, after the code that creates and populates the `empTree` binary tree:

```
static void doWork()
{
    ...
    Console.WriteLine("All employees");
    var allEmployees = from e in empTree
        select e;

    foreach (var emp in allEmployees)
    {
        Console.WriteLine(emp);
    }
    ...
}
```

This code generates an enumerable collection of employees named `allEmployees` and then iterates through this collection, displaying the details of each employee.

4. Add the following code immediately after the statements you typed in the previous step:

```
static void doWork()
{
    ...
    empTree.Insert(new Employee
    {
        Id = 7,
        FirstName = "David",
        LastName = "Simpson",
        Department = "IT"
    });

    Console.WriteLine();
    Console.WriteLine("Employee added");

    Console.WriteLine("All employees");
    foreach (var emp in allEmployees)
    {
        Console.WriteLine(emp);
    }
    ...
}
```

These statements add a new employee to the `empTree` tree and then iterate through the `allEmployees` collection again.

5. On the **Debug** menu, select **Start Without Debugging**. Then verify that the output of the application looks like this:

```
All employees
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 2, Name: Jeff Hay, Dept: Marketing
Id: 3, Name: Dave Barnett, Dept: Sales
Id: 4, Name: Charlie Herb, Dept: IT
Id: 5, Name: Tim Litton, Dept: Marketing
Id: 6, Name: Chris Preston, Dept: Sales
```

```
Employee added
All employees
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 2, Name: Jeff Hay, Dept: Marketing
Id: 3, Name: Dave Barnett, Dept: Sales
Id: 4, Name: Charlie Herb, Dept: IT
Id: 5, Name: Tim Litton, Dept: Marketing
Id: 6, Name: Chris Preston, Dept: Sales
Id: 7, Name: David Simpson, Dept: IT
```

Notice that the second time the application iterates through the `allEmployees` collection, the list displayed includes David Simpson, even though this employee was added only after the `allEmployees` collection was defined.

6. Press **Enter** to return to Visual Studio 2022.
7. In the `doWork` method, change the statement that generates the `allEmployees` collection to identify and cache the data immediately, as shown here in bold:

```
var allEmployees = from e in empTree.ToList<Employee>()
                    select e;
```

LINQ provides generic and nongeneric versions of the `ToList` and `ToArray` methods. If possible, it's better to use the generic versions of these methods to ensure the type safety of the result. The data returned by the `select` operator is an `Employee` object, and the code shown in this step generates `allEmployees` as a generic `List<Employee>` collection.

8. On the **Debug** menu, select **Start Without Debugging**. Then verify that the output of the application looks like this:

```
All employees
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 2, Name: Jeff Hay, Dept: Marketing
Id: 3, Name: Dave Barnett, Dept: Sales
Id: 4, Name: Charlie Herb, Dept: IT
Id: 5, Name: Tim Litton, Dept: Marketing
Id: 6, Name: Chris Preston, Dept: Sales
Employee added
All employees
Id: 1, Name: Kim Abercrombie, Dept: IT
Id: 2, Name: Jeff Hay, Dept: Marketing
Id: 3, Name: Dave Barnett, Dept: Sales
Id: 4, Name: Charlie Herb, Dept: IT
Id: 5, Name: Tim Litton, Dept: Marketing
Id: 6, Name: Chris Preston, Dept: Sales
```

Notice that the second time the application iterates through the `allEmployees` collection, the list displayed does not include David Simpson. In this case, the query is evaluated, and the results are cached before David Simpson is added to the `empTree` binary tree.

9. Press **Enter** to return to Visual Studio 2022.

Summary

In this chapter, you learned how LINQ uses the `IEnumerable<T>` interface and extension methods to provide a mechanism for querying data. You also saw how these features support the query expression syntax in C#.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 22, “Operator overloading.”
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

To	Do this
Project specified fields from an enumerable collection	Use the <code>Select</code> method and specify a lambda expression that identifies the fields to project. For example: <pre>var customerFirstNames = customers.Select(cust => cust.FirstName);</pre> Or use the <code>from</code> and <code>select</code> query operators. For example: <pre>var customerFirstNames = from cust in customers select cust.FirstName;</pre>
Filter rows from an enumerable collection	Use the <code>Where</code> method and specify a lambda expression containing the criteria that rows should match. For example: <pre>var usCompanies = addresses .Where(addr => String.Equals(addr.Country, "United States")) .Select(usComp => usComp.CompanyName);</pre> Or use the <code>where</code> query operator. For example: <pre>var usCompanies = addresses .Where(addr => String.Equals(addr.Country, "United States")) .Select(usComp => usComp.CompanyName);</pre>
Enumerate data in a specific order	Use the <code>OrderBy</code> method and specify a lambda expression identifying the field to use to order rows. For example: <pre>var companyNames = addresses .OrderBy(addr => addr.CompanyName) .Select(comp => comp.CompanyName);</pre> Or, use the <code>orderby</code> query operator. For example: <pre>var companyNames = from a in addresses orderby a.CompanyName select a.CompanyName;</pre>
Group data by the values in a field	Use the <code>GroupBy</code> method and specify a lambda expression identifying the field to use to group rows. For example: <pre>var companiesGroupedByCountry = addresses.GroupBy(addr => addr.Country);</pre> Or, use the <code>group by</code> query operator. For example: <pre>var companiesGroupedByCountry = from a in addresses group a by a.Country;</pre>

Join data held in two different collections	Use the Join method, specifying the collection with which to join, the join criteria, and the fields for the result. For example: <pre>var countriesAndCustomers = customers .Select(c => new { c.FirstName, c.LastName, c.CompanyName }) .Join(addresses, custs => custs.CompanyName, addr => addr.CompanyName, (custs, addr) => new {custs.FirstName, custs.LastName, addr.Country });</pre> Or, use the join query operator. For example: <pre>var countriesAndCustomers = from a in addresses join c in customers on a.CompanyName equals c.CompanyName select new { c.FirstName, c.LastName, a.Country };</pre>
Force immediate generation of the results for a LINQ query	Use the ToList or ToArray method to generate a list or an array containing the results. For example: <pre>var allEmployees = from e in empTree.ToList<Employee>() select e;</pre>

Operator overloading

After completing this chapter, you will be able to:

- Explain how operator overloading works.
- Write increment and decrement operators for your own types.
- Understand the nuances of operators in structs and classes.
- Explain the need to implement some operators as pairs.
- Override operators in a struct, and implement custom equality operators.
- Implement implicit and explicit conversion operators for your own types.

The examples throughout this book make great use of the standard operator symbols (such as `+` and `-`) to perform standard operations (such as addition and subtraction) on types (such as `int` and `double`). Many of the built-in types come with their own predefined behaviors for each operator. You can also define how operators should behave in your own structures and classes, which is the subject of this chapter.

Understanding operators

It's worth recapping some of the fundamental aspects of operators before delving into the details of how they work and how you can overload them. The following list summarizes these aspects:

- You use operators to combine operands into expressions. Each operator has its own semantics, which are dependent on the type it works with. For example, the addition (`+`) operator means *add* when you use it with numeric types or *concatenate* when you use it with strings.
- Each operator has a *precedence*. For example, the multiplication (`*`) operator has a higher precedence than the `+` operator. This means that the expression `a + b * c` is the same as `a + (b * c)`.

- Each operator has an *associativity* that defines whether it evaluates from left to right or from right to left. For example, the equals (=) operator is right-associative (it evaluates from right to left), so `a = b = c` is the same as `a = (b = c)`.
- A *unary operator* is an operator that has just one operand. For example, the increment operator (++) is a unary operator.
- A *binary operator* is an operator that has two operands. For example, the * operator is a binary operator.

Operator constraints

This book presents many examples of how you can overload methods when defining your own types. With C#, you can also overload many of the existing operator symbols for your own types, although the syntax is slightly different. When you do this, the operators you implement automatically fall into a well-defined framework with the following rules:

- You can't change the precedence or associativity of an operator. Precedence and associativity are based on the operator symbol (for example, +) and not on the type (for example, `int`) on which the operator symbol is being used. So, the expression `a + b * c` is always the same as `a + (b * c)` regardless of the types of `a`, `b`, and `c`.
- You can't change the multiplicity (the number of operands) of an operator. For example, * (the symbol for multiplication) is a binary operator. If you declare a * operator for your own type, it must be a binary operator.
- You can't invent new operator symbols. For example, you can't create an operator symbol such as ** for raising one number to the power of another number. You'd have to define a method to do that.
- You can't change the meaning of operators when they are applied to built-in types. For example, the expression `1 + 2` has a predefined meaning, and you're not allowed to override this meaning. If you could do this, things would be too complicated.
- You can't overload some operator symbols. For example, you can't overload the dot (.) operator, which indicates access to a class member. Again, if you could do this, it would lead to unnecessary complexity.



Tip You can use indexers to simulate [] as an operator. Similarly, you can use properties to simulate assignment (=) as an operator, and you can use delegates to mimic a function call as an operator.

Overloaded operators

To define your own operator behavior, you must overload a selected operator. You use method-like syntax with a return type and parameters, but the name of the method is the keyword `operator` together with the operator symbol that you're declaring. For example, the following code shows a user-defined structure named `Hour` that defines a binary `+` operator to add together two instances of `Hour`.

```
struct Hour
{
    public Hour(int initialValue) => this.value = initialValue;

    public static Hour operator +(Hour lhs, Hour rhs) => new Hour(lhs.value + rhs.value);
    ...
    private int value;
}
```

Notice the following:

- The operator is public. All operators *must* be public.
- The operator is static. All operators *must* be static. Operators are never polymorphic and cannot use the `virtual`, `abstract`, `override`, or `sealed` modifiers.
- A binary operator (such as the `+` operator shown in this example) has two explicit arguments, and a unary operator has one explicit argument. (C++ programmers should note that operators never have a hidden `this` parameter.)



Tip When you declare highly stylized functionality (such as operators), it's useful to adopt a naming convention for the parameters. For example, developers often use `lhs` and `rhs` (acronyms for left-hand side and right-hand side, respectively) for binary operators.

When you use the `+` operator on two expressions of type `Hour`, the C# compiler automatically converts your code to a call to your operator `+` method. The C# compiler transforms this code

```
Hour Example(Hour a, Hour b) => a + b;
```

to this:

```
Hour Example(Hour a, Hour b) => Hour.operator +(a,b); // pseudocode
```

Note, however, that this syntax is pseudocode and not valid C#. You can use a binary operator only in its standard infix notation (with the symbol between the operands).

There is one final rule that you must follow when declaring an operator: At least one parameter must be of the containing type. In the preceding operator `+` example for the `Hour` class, one of the parameters, `a` or `b`, must be an `Hour` object. In this example, both parameters are `Hour` objects. However, suppose, for example, that you want to define additional implementations of operator `+` that add an integer (a number of hours) to an `Hour` object. In that case, the first parameter could be

Hour, and the second parameter could be the integer. This rule makes it easier for the compiler to know where to look when trying to resolve an operator invocation, and it also ensures that you can't change the meaning of the built-in operators.

Creating symmetric operators

In the preceding section, you saw how to declare a binary `+` operator to add together two instances of type `Hour`. The `Hour` structure also has a constructor that creates an `Hour` from an `int`. This means that you can add together an `Hour` and an `int`; you just have to first use the `Hour` constructor to convert the `int` to an `Hour`, as in the following example:

```
Hour a = ...;
int b = ...;
Hour later = a + new Hour;
```

This is certainly valid code, but it's not as clear or concise as adding an `Hour` and an `int` directly, like this:

```
Hour a = ...;
int b = ...;
Hour later = a + b;
```

To make the expression `a + b` valid, you must specify what it means to add together an `Hour` (`a`, on the left) and an `int` (`b`, on the right). In other words, you must declare a binary `+` operator whose first parameter is an `Hour` and whose second parameter is an `int`. The following code shows the recommended approach:

```
struct Hour
{
    public Hour(int initialValue) => this.value = initialValue;
    ...
    public static Hour operator +(Hour lhs, Hour rhs) => new Hour(lhs.value + rhs.value);

    public static Hour operator +(Hour lhs, int rhs) => lhs + new Hour(rhs);
    ...
    private int value;
}
```

Notice that all the second version of the operator does is construct an `Hour` from its `int` argument and then call the first version. In this way, the real logic behind the operator is held in a single place. The point is that the extra operator `+` simply makes existing functionality easier to use. Also, notice that you should not provide many different versions of this operator, each with a different second parameter type; instead, cater to the common and meaningful cases only, and let the user of the class take any additional steps if an unusual case is required.

This operator `+` declares how to add together an `Hour` as the left operand and an `int` as the right operand. It does not declare how to add together an `int` as the left operand and an `Hour` as the right operand:

```
int a = ...;
Hour b = ...;
Hour later = a + b; // compile-time error
```

This is counterintuitive. If you can write the expression `a + b`, you expect also to be able to write `b + a`. Therefore, you should provide another overload of operator `+`:

```
struct Hour
{
    public Hour(int initialValue) => this.value = initialValue;
    ...
    public static Hour operator +(Hour lhs, int rhs) => lhs + new Hour(rhs);

    public static Hour operator +(int lhs, Hour rhs) => new Hour(lhs) + rhs;
    ...
    private int value;
}
```



Note C++ programmers should note that you must provide the overload yourself. The compiler won't write the overload for you or silently swap the sequence of the two operands to find a matching operator.

Understanding compound assignment evaluation

A compound assignment operator (such as `+=`) is always evaluated in terms of its associated simple operator (such as `+`). In other words, the statement

```
a += b;
```

is automatically evaluated like this:

```
a = a + b;
```

In general, the expression `a @= b` (where `@` represents any valid operator) is always evaluated as `a = a @ b`. If you've overloaded the appropriate simple operator, the overloaded version is automatically called when you use its associated compound assignment operator, as is shown in the following example:

```
Hour a = ...;
int b = ...;
a += a; // same as a = a + a
a += b; // same as a = a + b
```

The first compound assignment expression (`a += a`) is valid because `a` is of type `Hour`, and the `Hour` type declares a binary operator `+` whose parameters are both `Hour`. Similarly, the second compound assignment expression (`a += b`) is also valid because `a` is of type `Hour` and `b` is of type `int`. The `Hour` type also declares a binary operator `+` whose first parameter is an `Hour` and whose second parameter is an `int`. Be aware, however, that you cannot write the expression `b += a` because that's the same as `b = b + a`. Although the addition is valid, the assignment is not, because there's no way to assign an `Hour` to the built-in `int` type.

Declaring increment and decrement operators

With C#, you can declare your own version of the increment (++) and decrement (--) operators. The usual rules apply when declaring these operators: they must be public, they must be static, and they must be unary (they can take only a single parameter). Here is the increment operator for the Hour structure:

```
struct Hour
{
    ...
    public static Hour operator ++(Hour arg)
    {
        arg.value++;
        return arg;
    }
    ...

    private int value;
}
```

The increment and decrement operators are unique in that they can be used in prefix and postfix forms. C# cleverly uses the same single operator for both the prefix and postfix versions. The result of a postfix expression is the value of the operand *before* the expression takes place. In other words, the compiler effectively converts the code

```
Hour now = new Hour(9);
Hour postfix = now++;
```

to this:

```
Hour now = new Hour(9);
Hour postfix = now;
now = Hour.operator ++(now); // pseudocode, not valid C#
```

The result of a prefix expression is the return value of the operator, so the C# compiler effectively transforms the code

```
Hour now = new Hour(9);
Hour prefix = ++now;
```

to this:

```
Hour now = new Hour(9);
now = Hour.operator ++(now); // pseudocode, not valid in C#
Hour prefix = now;
```

This equivalence means that the return type of the increment and decrement operators must be the same as the parameter type.

Comparing operators in structures and classes

The implementation of the increment operator in the Hour structure works only because Hour is a structure. If you change Hour into a class but leave the implementation of its increment operator unchanged, the postfix translation won't give the correct answer. If you remember that a class is

a reference type and if you revisit the compiler translations explained earlier, you can see in the following example why the operators for the `Hour` class no longer function as expected:

```
Hour now = new Hour(9);
Hour postfix = now;
now = Hour.operator ++(now); // pseudocode, not valid C#
```

If `Hour` is a class, the assignment statement `postfix = now` makes the variable `postfix` refer to the same object as `now`. Updating `now` automatically updates `postfix`! If `Hour` is a structure, the assignment statement makes a copy of `now` in `postfix`, and any changes to `now` leave `postfix` unchanged, which is what you want.

The correct implementation of the increment operator when `Hour` is a class is as follows:

```
class Hour
{
    public Hour(int initialValue) => this.value = initialValue;
    ...
    public static Hour operator ++(Hour arg) => new Hour(arg.value + 1);
    ...
    private int value;
}
```

Notice that operator `++` now creates a new object based on the data in the original. The data in the new object is incremented, but the data in the original is left unchanged. Although this works, the compiler translation of the increment operator results in a new object being created each time it's used. This can be expensive in terms of memory use and garbage-collection overhead. Therefore, it's recommended that you limit operator overloads when you define types. This recommendation applies to all operators, not just to the increment operator.

Defining operator pairs

Some operators naturally come in pairs. For example, if you can compare two `Hour` values by using the `!=` operator, you would expect to be able to also compare two `Hour` values by using the `==` operator. The C# compiler enforces this very reasonable expectation by insisting that if you define either operator `==` or operator `!=`, you must define them both. This neither-or-both rule also applies to the `<` and `>` operators and the `<=` and `>=` operators. The C# compiler does not write any of these operator partners for you. You must write them all explicitly yourself, regardless of how obvious they might seem. Here are the `==` and `!=` operators for the `Hour` structure:

```
struct Hour
{
    public Hour(int initialValue) => this.value = initialValue;
    ...
    public static bool operator ==(Hour lhs, Hour rhs) => lhs.value == rhs.value;
    public static bool operator !=(Hour lhs, Hour rhs) => lhs.value != rhs.value;
    ...
    private int value;
}
```

The return type from these operators doesn't actually have to be Boolean. However, you should have a very good reason for using some other type, or these operators could become very confusing.

Implementing operators

A complex number has two elements: a real component and an imaginary component. Typically, a complex number is represented in the form $(x + iy)$, where x is the real component, and iy is the imaginary component. The values of x and y are regular integers, and i represents the square root of -1 (which is why iy is imaginary).

Despite their rather obscure and theoretical feel, complex numbers have numerous uses in electronics, applied mathematics, and physics, and in many aspects of engineering. If you want more information about how and why complex numbers are useful, Wikipedia provides a useful and informative article; search for "Complex number."



Note Microsoft .NET version 4.0 and later includes a struct called `Complex` in the `System.Numerics` namespace that implements complex numbers, so there's no real need to define your own version of this type anymore. However, it's still instructive to see how to implement some of the common operators for this type.

In the following exercise, you'll develop a type that simulates complex numbers. You'll implement complex numbers as a pair of integers that represent the coefficients x and y for the real and imaginary elements. You will also implement the operators necessary for performing simple arithmetic using complex numbers. The following table summarizes how to perform the four primary arithmetic operations on a pair of complex numbers, $(a + bi)$ and $(c + di)$:

Operation	Calculation
$(a + bi) + (c + di)$	$((a + c) + (b + d)i)$
$(a + bi) - (c + di)$	$((a - c) + (b - d)i)$
$(a + bi) * (c + di)$	$((a * c - b * d) + (b * c + a * d)i)$
$(a + bi) / (c + di)$	$(((a * c + b * d) / (c * c + d * d)) + ((b * c - a * d) / (c * c + d * d))i)$

To create the Complex struct and implement the arithmetic operators

1. Start Visual Studio 2022, if it is not already running.
2. Open the **ComplexNumbers** solution, which is located in the `\Microsoft Press\VCSBS\Chapter 22\ComplexNumbers` folder in your **Documents** folder. This is a console application that you will use to build and test your code. The `Program.cs` file contains the familiar `doWork` method.

3. In Solution Explorer, select the **ComplexNumbers** project.
4. On the **Project** menu, select **Add Class**.
5. In the Add New Item – ComplexNumbers dialog, in the **Name** box, type **Complex.cs**. Then select **Add**.

Visual Studio creates the `Complex` struct and opens the `Complex.cs` file in the Code and Text Editor window.

6. Add the automatic integer properties `Real` and `Imaginary` to the `Complex` struct, as shown in the code in bold that follows:

```
struct Complex
{
    public int Real { get; set; }
    public int Imaginary { get; set; }
}
```

These properties will hold the real and imaginary components of a complex number.

7. Add the constructor shown here in bold to the `Complex` struct:

```
struct Complex
{
    ...
    public Complex (int real, int imaginary)
    {
        this.Real = real;
        this.Imaginary = imaginary;
    }
}
```

This constructor takes two `int` parameters and uses them to populate the `Real` and `Imaginary` properties.

8. Override the `ToString` method as shown next in bold:

```
struct Complex
{
    ...
    public override string ToString() => $"{this.Real} + {this.Imaginary}i ";
}
```

This method returns a string representing the complex number in the form $(x + yi)$.

9. Add the overloaded `+` operator to the `Complex` struct as shown in bold in the code that follows:

```
struct Complex
{
    ...
    public static Complex operator +(Complex lhs, Complex rhs) =>
        new Complex(lhs.Real + rhs.Real, lhs.Imaginary + rhs.Imaginary);
}
```

This is the binary addition operator. It adds two `Complex` objects together by performing the calculation shown in the table preceding the exercise. The operator returns a new `Complex` object containing the results of this calculation.

10. Add the overloaded `-` operator to the `Complex` struct.

```
struct Complex
{
    ...
    public static Complex operator -(Complex lhs, Complex rhs) =>
        new Complex(lhs.Real - rhs.Real, lhs.Imaginary - rhs.Imaginary);
}
```

This operator follows the same form as the overloaded `+` operator.

11. Implement the `*` operator and `/` operators by adding the code shown in bold to the `Complex` struct.

```
struct Complex
{
    ...
    public static Complex operator *(Complex lhs, Complex rhs) =>
        new Complex(lhs.Real * rhs.Real - lhs.Imaginary * rhs.Imaginary,
            lhs.Imaginary * rhs.Real + lhs.Real * rhs.Imaginary);

    public static Complex operator /(Complex lhs, Complex rhs)
    {
        int realElement = (lhs.Real * rhs.Real + lhs.Imaginary * rhs.Imaginary) /
            (rhs.Real * rhs.Real + rhs.Imaginary * rhs.Imaginary);

        int imaginaryElement = (lhs.Imaginary * rhs.Real - lhs.Real * rhs.Imaginary) /
            (rhs.Real * rhs.Real + rhs.Imaginary * rhs.Imaginary);

        return new Complex(realElement, imaginaryElement);
    }
}
```

These operators follow the same form as the previous two operators, although the calculations are a little more complicated. (The calculation for the `/` operator has been broken down into two steps to avoid lengthy lines of code.)

12. Display the `Program.cs` file in the Code and Text Editor window. Then add the following statements shown in bold to the `doWork` method of the `Program` class and delete the `// TODO: comment`:

```
static void doWork()
{
    Complex first = new Complex(10, 4);
    Complex second = new Complex(5, 2);
    Console.WriteLine($"first is {first}");
    Console.WriteLine($"second is {second}");

    Complex temp = first + second;
    Console.WriteLine($"Add: result is {temp}");
}
```

```

temp = first - second;
Console.WriteLine($"Subtract: result is {temp}");

temp = first * second;
Console.WriteLine($"Multiply: result is {temp}");

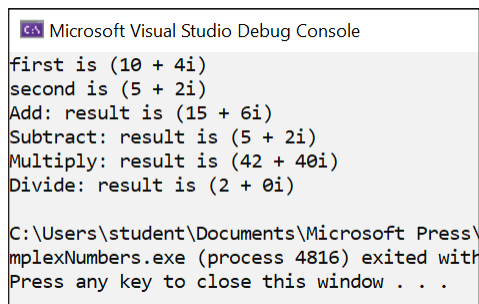
temp = first / second;
Console.WriteLine($"Divide: result is {temp}");
}

```

This code creates two `Complex` objects that represent the complex values $(10 + 4i)$ and $(5 + 2i)$. The code displays them and then tests each of the operators you just defined, displaying the results in each case.

13. On the **Debug** menu, select **Start Without Debugging**.

Verify that the application displays the results shown in the following image:



```

Microsoft Visual Studio Debug Console
first is (10 + 4i)
second is (5 + 2i)
Add: result is (15 + 6i)
Subtract: result is (5 + 2i)
Multiply: result is (42 + 40i)
Divide: result is (2 + 0i)

C:\Users\student\Documents\Microsoft Press\
mplexNumbers.exe (process 4816) exited with
Press any key to close this window . . .

```

14. Close the application, and return to the Visual Studio 2022 programming environment.

You have now created a type that models complex numbers and supports basic arithmetic operations.

Overriding the equality operators

The default implementation of the equality operators for a class looks at the address of the class in memory. Two instances of a class are considered equal if they have the same address—that is, they reference the same object. If you want to test for equality based on the contents of the fields in a class, you can override the `==` and `!=` operators.

For a struct, the `==` and `!=` operators examine the contents of the fields. Two instances of a struct are considered equal if their fields have the same value. This is one of the fundamental differences between a reference type (such as a class) and a value type.

The default implementation of the `==` and `!=` operators for a struct uses a generalized technique known as *reflection* to discover the fields so it can compare their contents. While this process will work for any struct, and the implementation has been refined over various iterations of .NET, it can be slow—especially if you’re comparing many thousands of structs in a collection. For this reason, you may want to provide your own custom, optimized version of the `==` and `!=` operators.

In both cases (for a struct and a class), if you define operator `==` and operator `!=`, you should also override the `Equals` and `GetHashCode` methods inherited from `System.Object` (or `System.ValueType` if you are creating a structure). The `Equals` method should exhibit the same behavior as operator `==`. (You should define one in terms of the other.) The `GetHashCode` method is used by other classes in the Microsoft .NET libraries. When you use an object as a key in a hash table, for example, the `GetHashCode` method is called on the object to help calculate a hash value. (For more information, see the .NET reference documentation supplied with Visual Studio 2022.) All this method needs to do is return a distinguishing integer value. Don't return the same integer from the `GetHashCode` method of all your objects, however, because this will nullify the effectiveness of the hashing algorithms.

Finally, if you're creating a record type, don't override the default implementations of operator `==` and operator `!=`. Records automatically implement value equality; this is one of the main reasons for using a record. Consider defining a record instead of a struct when your type models data and should implement value equality.

In the next exercise, you'll extend the `Complex` struct and provide the equality operators `==` and `!=`.

To implement the equality operators

1. In Visual Studio 2022, display the `Complex.cs` file in the Code and Text Editor window.
2. Add the `==` and `!=` operators to the `Complex` struct as shown in bold in the following example:

```
struct Complex
{
    ...
    public static bool operator ==(Complex lhs, Complex rhs) => lhs.Equals(rhs);

    public static bool operator !=(Complex lhs, Complex rhs) => !lhs.Equals(rhs);
}
```

Notice that both of these operators use the `Equals` method. This method compares an instance of a class against another instance specified as an argument. It returns `true` if they have equivalent values and `false` otherwise. As described, you must provide your own implementation of this method for the equality operators to work as expected.

3. Override the `Equals` method in the `Complex` struct by adding the following code shown in bold:

```
struct Complex
{
    ...
    public override bool Equals(Object obj)
    {
        if (obj is Complex)
        {
```

```

        Complex compare = (Complex)obj;
        return (this.Real == compare.Real) &&
            (this.Imaginary == compare.Imaginary);
    }
    else
    {
        return false;
    }
}
}

```

The `Equals` method takes an `Object` as a parameter. This code verifies that the type of the parameter is actually a `Complex` object. If it is, this code compares the values in the `Real` and `Imaginary` properties in the current instance and the parameter passed in. If they're the same, the method returns `true`; otherwise, it returns `false`. If the parameter passed in is not a `Complex` object, the method returns `false`.

4. On the **Build** menu, select **Rebuild Solution**.

The Error List window displays the following warning messages:

```

'ComplexNumbers.Complex' overrides Object.Equals(object o) but does not override
Object.GetHashCode()
'ComplexNumbers.Complex' defines operator == or operator != but does not override
Object.GetHashCode()

```

If you define the `!=` and `==` operators, you should also override the `GetHashCode` methods inherited from `System.Object`.



Note If the Error List window is not visible, select Error List on the View menu.

5. Override the `GetHashCode` method to the `Complex` struct by adding the following code shown in bold. This implementation simply calls the method inherited from the `Object` class, but you can provide your own mechanism to generate a hash code for an object if you prefer.

```

Struct Complex
{
    ...
    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
}

```

6. On the **Build** menu, select **Rebuild Solution**.

Verify that the solution now builds without reporting any warnings.

7. Display the Program.cs file in the Code and Text Editor window. Then add the following code shown in bold to the end of the doWork method:

```
static void doWork()
{
    ...
    if (temp == first)
    {
        Console.WriteLine("Comparison: temp == first");
    }
    else
    {
        Console.WriteLine("Comparison: temp != first");
    }

    if (temp == temp)
    {
        Console.WriteLine("Comparison: temp == temp");
    }
    else
    {
        Console.WriteLine("Comparison: temp != temp");
    }
}
```



Note The expression `temp == temp` generates the warning message "Comparison made to same variable; did you mean to compare to something else?" In this case, you can ignore the warning because this comparison is intentional; it is to verify that the `==` operator is working as expected.

8. On the **Debug** menu, select **Start Without Debugging**. Then verify that the final two messages displayed are these:

```
Comparison: temp != first
Comparison: temp == temp
```

9. Close the application, and return to Visual Studio 2022.

Understanding conversion operators

Sometimes, you need to convert an expression of one type to another. For example, the following method is declared with a single `double` parameter:

```
class Example
{
    public static void MyDoubleMethod(double parameter)
    {
        ...
    }
}
```

You might reasonably expect that only values of type `double` could be used as arguments when your code calls `MyDoubleMethod`, but this is not so. The C# compiler also allows `MyDoubleMethod` to be called with an argument of some other type, but only if the value of the argument can be converted to a `double`. For example, if you provide an `int` argument, the compiler generates code that converts the value of the argument to a `double` when the method is called.

Providing built-in conversions

The built-in types have some built-in conversions. For example, as mentioned, an `int` can be implicitly converted to a `double`. An implicit conversion requires no special syntax and never throws an exception.

```
Example.MyDoubleMethod(42); // implicit int-to-double conversion
```

An implicit conversion is sometimes called a *widening conversion* because the result is *wider* than the original value—it contains at least as much information as the original value, and nothing is lost. In the case of `int` and `double`, the range of `double` is greater than that of `int`, and all `int` values have an equivalent `double` value. However, the converse is not true, and a `double` value cannot be implicitly converted to an `int`:

```
class Example
{
    public static void MyIntMethod(int parameter)
    {
        ...
    }
}
...
Example.MyIntMethod(42.0); // compile-time error
```

When you convert a `double` to an `int`, you run the risk of losing information, so the conversion will not be performed automatically. (Consider what would happen if the argument to `MyIntMethod` were 42.5. How should this be converted?) A `double` can be converted to an `int`, but the conversion requires an explicit notation (a cast):

```
Example.MyIntMethod((int)42.0);
```

An explicit conversion is sometimes called a *narrowing conversion* because the result is *narrower* than the original value (that is, it can contain less information) and can throw an `OverflowException` if the resulting value is out of the range of the target type. In C#, you can create conversion operators for your own user-defined types to control whether it is sensible to convert values to other types, and you can also specify whether these conversions are implicit or explicit.

Implementing user-defined conversion operators

The syntax for declaring a user-defined conversion operator is similar to that for declaring an overloaded operator, but there are some important differences. Here's a conversion operator that allows an `Hour` object to be implicitly converted to an `int`:

```
struct Hour
{
    ...
    public static implicit operator int (Hour from)
    {
        return from.value;
    }

    private int value;
}
```

A conversion operator must be *public*, and it must also be *static*. The type you're converting from is declared as the parameter (in this case, `Hour`), and the type you're converting to is declared as the type name after the keyword `operator` (in this case, `int`). There is no return type specified before the keyword `operator`.

When declaring your own conversion operators, you must specify whether they are implicit conversion operators or explicit conversion operators. You do this by using the `implicit` and `explicit` keywords. The `Hour` to `int` conversion operator shown in the preceding example is implicit, meaning that the C# compiler can use it without requiring a cast.

```
class Example
{
    public static void MyOtherMethod(int parameter) { ... }
    public static void Main()
    {
        Hour lunch = new Hour(12);
        Example.MyOtherMethod(lunch); // implicit Hour to int conversion
    }
}
```

If the conversion operator had been declared with `explicit`, the preceding example would not have compiled because an explicit conversion operator requires a cast.

```
Example.MyOtherMethod((int)lunch); // explicit Hour to int conversion
```

When should you declare a conversion operator as explicit or implicit? If a conversion is always safe, does not run the risk of losing information, and cannot throw an exception, it can be defined as an implicit conversion. Otherwise, it should be declared as an explicit conversion. Converting from an `Hour` to an `int` is always safe—every `Hour` has a corresponding `int` value—so it makes sense for it to be implicit. An operator that converts a `string` to an `Hour` should be explicit because not all strings represent valid `Hours`. (The string `7` is fine, but how would you convert the string `Hello, World` to an `Hour`?)

Creating symmetric operators, revisited

Conversion operators provide an alternative way to resolve the problem of providing symmetric operators. For example, instead of providing three versions of operator `+` (`Hour + Hour`, `Hour + int`, and `int + Hour`) for the `Hour` structure, as shown earlier, you can provide a single version of operator `+` (that takes two `Hour` parameters) and an implicit `int` to `Hour` conversion, like this:

```
struct Hour
{
    public Hour(int initialValue) => this.value = initialValue;

    public static Hour operator +(Hour lhs, Hour rhs) => new Hour(lhs.value + rhs.value);

    public static implicit operator Hour (int from) => new Hour (from);
    ...
    private int value;
}
```

If you add an `Hour` to an `int` (in either order), the C# compiler automatically converts the `int` to an `Hour` and then calls operator `+` with two `Hour` arguments, as demonstrated here:

```
void Example(Hour a, int b)
{
    Hour eg1 = a + b; // b converted to an Hour
    Hour eg2 = b + a; // b converted to an Hour
}
```

Writing conversion operators

In the final exercise of this chapter, you'll add conversion operators to the `Complex` struct. You'll start by writing a pair of conversion operators that convert between the `int` type and the `Complex` type. Converting an `int` to a `Complex` object is always a safe process and never loses information (because an `int` is really just a complex number without an imaginary element). You will implement this as an implicit conversion operator. However, the converse is not true—to convert a `Complex` object into an `int`, you must discard the imaginary element. So, you'll implement this conversion operator as explicit.

To implement the conversion operators

1. Return to Visual Studio 2022 and display the `Complex.cs` file in the Code and Text Editor window.
2. Add the constructor shown in bold in the code that follows to the `Complex` struct, immediately after the existing constructor and before the `ToString` method. This new constructor takes a single `int` parameter, which it uses to initialize the `Real` property. The `Imaginary` property is set to 0.

```
struct Complex
{
    ...
    public Complex(int real)
    {
```

```

        this.Real = real;
        this.Imaginary = 0;
    }
    ...
}

```

3. Add the following implicit conversion operator shown in bold to the `Complex` struct:

```

struct Complex
{
    ...
    public static implicit operator Complex(int from) => new Complex(from);
    ...
}

```

This operator converts from an `int` to a `Complex` object by returning a new instance of the `Complex` struct built using the constructor you created in the previous step.

4. Add the following explicit conversion operator shown in bold to the `Complex` struct:

```

struct Complex
{
    ...
    public static explicit operator int(Complex from) => from.Real;
    ...
}

```

This operator takes a `Complex` object and returns the value of the `Real` property. This conversion discards the imaginary element of the complex number.

5. Display the `Program.cs` file in the Code and Text Editor window. Then add the following code shown in bold to the end of the `doWork` method:

```

static void doWork()
{
    ...
    Console.WriteLine($"Current value of temp is {temp}");

    if (temp == 2)
    {
        Console.WriteLine("Comparison after conversion: temp == 2");
    }
    else
    {
        Console.WriteLine("Comparison after conversion: temp != 2");
    }

    temp += 2;
    Console.WriteLine($"Value after adding 2: temp = {temp}");
}

```

These statements test the implicit operator that converts an `int` to a `Complex` object. The `if` statement compares a `Complex` object to an `int`. The compiler generates code that converts the `int` into a `Complex` object first and then invokes the `==` operator of the `Complex` struct. The statement that adds 2 to the `temp` variable converts the `int` value 2 into a `Complex` object and then uses the `+` operator of the `Complex` struct.

6. Add the following statements shown in bold to end of the `doWork` method:

```
static void doWork()
{
    ...
    int tempInt = temp;
    Console.WriteLine($"Int value after conversion: tempInt == {tempInt}");
}
```

The first statement attempts to assign a `Complex` object to an `int` variable. The second statement displays the result.

7. On the **Build** menu, select **Rebuild Solution**.

The solution fails to build, and the compiler reports the following error in the Error List window:

Cannot implicitly convert type 'ComplexNumbers.Complex' to 'int'. An explicit conversion exists (are you missing a cast?)

The operator that converts from a `Complex` object to an `int` is an explicit conversion operator, so you must specify a cast.

8. Modify the statement that attempts to store a `Complex` value in an `int` variable to use a cast, like this:

```
int tempInt = (int)temp;
```

9. On the **Debug** menu, select **Start Without Debugging**. Then verify that the solution builds and that the final four messages displayed look like this:

```
Current value of temp is (2 + 0i)
Comparison after conversion: temp == 2
Value after adding 2: temp = (4 + 0i)
Int value after conversion: tempInt == 4
```

10. Close the application, and return to Visual Studio 2022.

Summary

In this chapter, you learned how to overload operators and provide functionality specific to a class or structure. You implemented several common arithmetic operators, and you created operators with which you can compare instances of a class. Finally, you learned how to create implicit and explicit conversion operators.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 23, "Improving throughput by using tasks."
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

To	Do this
Implement an operator	Write the keywords <code>public</code> and <code>static</code>, followed by the return type, followed by the operator keyword, followed by the operator symbol being declared, followed by the appropriate parameters between parentheses. Implement the logic for the operator in the body of the method. For example: <pre>struct Complex { ... public static bool operator==(Complex lhs, Complex rhs) { ... // Implement logic for == operator } ... }</pre>
Define a conversion operator	Write the keywords <code>public</code> and <code>static</code>, followed by the keyword <code>implicit</code> or <code>explicit</code>, followed by the operator keyword, followed by the type the data is being converted to, followed by the type the data is being converted from as a single parameter between parentheses. For example: <pre>struct Complex { ... public static implicit operator Complex(int from) { ... // code to convert from an int } ... }</pre>

PART IV

Building Universal Windows Platform applications with C#

CHAPTER 23	Improving throughput by using tasks	559
CHAPTER 24	Improving response time by performing asynchronous operations.....	599
CHAPTER 25	Implementing the user interface for a Universal Windows Platform app.....	641
CHAPTER 26	Displaying and searching for data in a Universal Windows Platform app.....	687
CHAPTER 27	Accessing a remote database from a Universal Windows Platform app.....	717

So far, you've gained a thorough grounding in the syntax and semantics of the C# language. It's now time to examine how you can use this knowledge to take advantage of the features that Windows 11 provides for building applications that run unchanged on devices ranging from a desktop PC to a smartphone. You can construct applications that run in different environments by using the Universal Windows Platform (UWP) application framework. UWP applications can detect and adapt to the hardware on which they execute. They can receive input through a touch-sensitive screen or by using voice commands, and a UWP app can be designed to be aware of the location and orientation of the device it's running on. You can also build cloud-connected applications; these applications are not tied to a specific computer but can follow users when they sign in on another device. In short, Visual Studio 2022 provides the tools for developing highly mobile, highly graphical, highly connected applications that can run almost anywhere.

Part IV introduces you to the requirements of building UWP applications. You'll see examples of the asynchronous model of programming developed as part of .NET. You'll also learn how to build a UWP application that connects to the cloud to retrieve and present complex information in a natural and easily navigable style.

Improving throughput by using tasks

After completing this chapter, you will be able to:

- Describe the benefits of implementing parallel operations in an application.
- Explain the relationship between parallelization and the number of processor cores available.
- Use the `Task` class and the `Parallel` class to create and run parallel operations in an application.
- Cancel long-running tasks and handle exceptions raised by parallel operations.

In the bulk of this book, you've learned how to use C# to write programs that run in a single-threaded manner. By *single-threaded*, I mean that at any one point in time, a program executes a single instruction. This might not always be the most efficient approach for an application to take, however. If you have the appropriate processing resources available, some applications might run more quickly if you divide them into parallel paths of execution that can run concurrently.

This chapter is concerned with improving throughput in your applications by maximizing the use of available processing power. Specifically, you'll learn how to use `Task` objects to apply effective multitasking to computationally intensive applications.

Why perform multitasking by using parallel processing?

There are two primary reasons why you might want to perform multitasking in an application:

- **To improve responsiveness** A long-running operation might involve tasks that don't require processor time. Common examples include I/O-bound operations such as reading from or writing to a local disk or sending and receiving data across a network. In both of these cases, it doesn't make sense to have a program burn CPU cycles waiting for the operation to complete when the program could be doing something more useful instead, such as responding to user input. Most mobile-device users take this form of responsiveness for granted and don't expect their tablet to simply halt while it's sending and receiving email, for example. Chapter 24, "Improving response time by performing asynchronous operations," discusses this topic in more detail.

- **To improve scalability** If an operation is CPU bound, you can improve scalability by making efficient use of available processing resources and using these resources to reduce the time required to execute the operation. A developer can determine which operations include tasks that can be performed in parallel and arrange for these elements to run concurrently. As more computing resources are added, more instances of these tasks can be run in parallel. Until relatively recently, this model was suitable only for scientific and engineering systems that either had multiple CPUs or could spread the processing across different computers networked together. However, most modern computing devices now contain powerful CPUs that are capable of supporting true multitasking, and many operating systems provide primitives that enable you to parallelize tasks quite easily.

The rise of the multicore processor

At the turn of the 21st century, the cost of a decent personal computer was in the range of \$800 to \$1,500. Today, a decent personal computer still costs about the same—even after 2 decades of price inflation.

The specification of a typical computer these days is likely to include a processor running at a speed of between 3 GHz and 5 GHz, more than 1,000 GB of hard disk storage (possibly using solid-state technology, for speed), 8 GB to 16 GB of RAM, high-speed and high-resolution graphics, fast network interfaces, and a rewritable DVD drive. In contrast, the processor speed for a typical machine 17 years ago was between 500 MHz and 1 GHz, 80 GB was a large hard disk, Windows ran quite happily with 256 MB or less of RAM, rewritable CD drives cost well over \$100, and rewritable DVD drives were rare and extremely expensive. This is the joy of technological progress: ever-faster and more-powerful hardware at cheaper and cheaper prices.

This is not a new trend. In 1965, Gordon E. Moore, cofounder of Intel, wrote a paper titled “Cramming More Components onto Integrated Circuits,” which discussed how the increasing miniaturization of components enabled more transistors to be embedded on a silicon chip, and how, as the technology became more accessible and the costs of production fell, economics would dictate the need to squeeze as many as 65,000 components onto a single chip by 1975. Moore’s observations led to the dictum frequently referred to as *Moore’s Law*, which states that the number of transistors that can be placed inexpensively on an integrated circuit will increase exponentially, doubling approximately every two years. (Actually, Moore was initially more optimistic than this, postulating that the volume of transistors was likely to double every year, but he later modified his calculations.)

The ability to pack transistors together led to the ability to pass data between them more quickly. This meant we could expect to see chip manufacturers produce faster and more powerful microprocessors at an almost unrelenting pace, enabling software developers to write ever more complicated software that would run more quickly.

Moore’s Law concerning the miniaturization of electronic components still holds, even half a century later. However, physics has started to intervene. A limit occurs when it is not possible to transmit signals between transistors on a single chip any faster, no matter how small or densely packed they are. To a software developer, the most noticeable result of this limitation is that processors have stopped

getting faster at the same rate. Fifteen years ago, a fast processor ran at 3 GHz. Today, a fast processor runs at 5 GHz. This is still a sizable increase in speed, but not to the same scale as Moore's Law.

The limit to the speed at which processors can transmit data between components has caused chip companies to look at alternative mechanisms for increasing the amount of work a processor can do. The result is that most modern processors now have two or more *processor cores*. Effectively, chip manufacturers have put multiple processors on the same chip and added the necessary logic to enable them to communicate and coordinate with one another. Quad-core (four cores) and eight-core processors are now common, and chips with 16, 32, and 64 cores are also available. Moreover, the price of dual-core and quad-core processors is now sufficiently low that they are an expected element in laptops, tablets, and smartphones. So, although processors have stopped speeding up, you can now expect to get more of them on a single chip.

What does this mean to a C# developer? In the days before multicore processors, you could speed up a single-threaded application simply by running it on a faster processor. With multicore processors, this is no longer the case. A single-threaded application will run at the same speed on a single-core, dual-core, or quad-core processor that all have the same clock frequency. The difference is that on a dual-core processor, as far as your application is concerned, one of the processor cores will be sitting around idle, and on a quad-core processor, three of the cores will be simply ticking away, waiting for work. To make the best use of multicore processors, you must write your applications to take advantage of multitasking.

Implementing multitasking by using Microsoft .NET

Multitasking is the ability to do more than one thing at the same time. It is one of those concepts that's easy to describe but until recently has been difficult to implement.

In the optimal scenario, an application running on a multicore processor performs as many concurrent tasks as there are processor cores available, keeping each of the cores busy. However, to implement concurrency, you must consider many issues, including the following:

- How can you divide an application into a set of concurrent operations?
- How can you arrange for a set of operations to execute concurrently, on multiple processors?
- How can you ensure that you attempt to perform only as many concurrent operations as there are processors available?
- If an operation is blocked (such as while waiting for I/O to complete), how can you detect this and arrange for the processor to run a different operation rather than sit idle?
- How can you determine when one or more concurrent operations have completed?

To an application developer, the first question is a matter of application design. The remaining questions depend on the programmatic infrastructure. Microsoft provides the `Task` class and a collection of associated types in the `System.Threading.Tasks` namespace to help address these issues.



Important The point about application design is fundamental. If an application is not designed with multitasking in mind, then it doesn't matter how many processor cores you throw at it; it won't run any faster than it would on a single-core machine.

Tasks, threads, and the ThreadPool

The Task class is an abstraction of a concurrent operation. You create a Task object to run a block of code. You can instantiate multiple Task objects and run them in parallel if sufficient processors or processor cores are available.



Note From now on, I'll use the term *processor* to refer to either a single-core processor or a single processor core on a multicore processor.

Internally, the Windows Runtime (WinRT) implements tasks and schedules them for execution by using Thread objects and the ThreadPool class. Multithreading and thread pools have been available as part of .NET since version 1.0. If you're building traditional desktop applications, you can use the Thread class in the System.Threading namespace directly in your code. However, the Thread class is not available for Universal Windows Platform (UWP) apps. Instead, you use the Task class.

The Task class provides a powerful abstraction for threading with which you can easily distinguish between the degree of parallelization in an application (the tasks) and the units of parallelization (the threads). On a single-processor computer, these items are usually the same. However, on a computer with multiple processors or with a multicore processor, they are different.

If you design a program based directly on threads, you'll find that your application might not scale very well. The program will use the number of threads you create explicitly, and the operating system will schedule only that number of threads. This can lead to overloading and poor response time if the number of threads greatly exceeds the number of available processors. It can also lead to inefficiency and poor throughput if the number of threads is less than the number of processors.

WinRT optimizes the number of threads required to implement a set of concurrent tasks and schedules them efficiently according to the number of available processors. It implements a queuing mechanism to distribute the workload across a set of threads allocated to a thread pool (implemented by using a ThreadPool object). When a program creates a Task object, the task is added to a global queue. When a thread becomes available, the task is removed from the global queue and is executed by that thread. The ThreadPool class implements a number of optimizations and uses a work-stealing algorithm to ensure that threads are scheduled efficiently.

The number of threads created to handle your tasks is not necessarily the same as the number of processors. Depending on the nature of the workload, one or more processors might be busy performing high-priority work for other applications and services. Consequently, the optimal number of threads for your application might be less than the number of processors in the machine. Alternatively,

one or more threads in an application might be waiting for long-running memory access, I/O, or network operation to complete, leaving the corresponding processors free. In this case, the optimal number of threads might be more than the number of available processors. WinRT follows an iterative strategy, known as a *hill-climbing algorithm*, to dynamically determine the ideal number of threads for the current workload.

The important point is that all you have to do in your code is divide, or *partition*, your application into tasks that can be run in parallel. WinRT takes responsibility for creating the appropriate number of threads based on your computer's processor architecture and workload, associating your tasks with these threads and arranging for them to be run efficiently.



Note It doesn't matter if you partition your work into too many tasks because WinRT will attempt to run only as many concurrent threads as is practical. In fact, you're encouraged to *overpartition* your work because this will help ensure that your application scales if you move it to a computer that has more processors available.

Creating, running, and controlling tasks

You can create Task objects by using the Task constructor. This constructor is overloaded, but all versions expect you to provide an `Action` delegate as a parameter. Chapter 20, "Decoupling application logic and handling events," illustrated that an `Action` delegate references a method that does not return a value. A Task object invokes this delegate when it is scheduled to run. The following example creates a Task object that uses a delegate to run the method called `doWork`:

```
Task task = new Task(doWork);
...
private void doWork()
{
    // The task runs this code when it is started
    ...
}
```



Tip The default `Action` type references a method that takes no parameters. Other overloads of the Task constructor take an `Action<object>` parameter representing a delegate that refers to a method that takes a single object parameter. With these overloads, you can pass data into the method run by the task. The following code shows an example:

```
Action<object> action;
action = doWorkWithObject;
object parameterData = ...;
Task task = new Task(action, parameterData);
...
private void doWorkWithObject(object o)
{
    ...
}
```


After you create a Task object, you can run it by using the Start method, like this:

```
Task task = new Task(...);  
task.Start();
```

The Start method is overloaded, and you can optionally specify a TaskCreationOptions object to provide hints about how to schedule and run the task.



More Info For more information about the TaskCreationOptions enumeration, consult the documentation describing the .NET class library that is provided with Visual Studio.

Creating and running a task is a very common process, and the Task class provides the static Run method with which you can combine these operations. The Run method takes an Action delegate specifying the operation to perform (like the Task constructor) but runs the task immediately. It returns a reference to the Task object. You can use it like this:

```
Task task = Task.Run(() => doWork());
```

When the method run by the task completes, the task finishes, and the thread used to run the task can be recycled to execute another task.

When a task completes, you can arrange for another task to be scheduled immediately by creating a *continuation*. To do this, you call the ContinueWith method of a Task object. When the action performed by the Task object completes, the scheduler automatically creates a new Task object to run the action specified by the ContinueWith method. The method specified by the continuation expects a Task parameter, and the scheduler passes into the method a reference to the task that completed. The value returned by ContinueWith is a reference to the new Task object. The following code example creates a Task object that runs the doWork method and specifies a continuation that runs the doMoreWork method in a new task when the first task completes:

```
Task task = new Task(doWork);  
task.Start();  
Task newTask = task.ContinueWith(doMoreWork);  
...  
private void doWork()  
{  
    // The task runs this code when it is started  
    ...  
}  
...  
private void doMoreWork(Task task)  
{  
    // The continuation runs this code when doWork completes  
    ...  
}
```

The `ContinueWith` method is heavily overloaded, and you can provide parameters that specify additional items, including a `TaskContinuationOptions` value. The `TaskContinuationOptions` type is an enumeration that contains a superset of the values in the `TaskCreationOptions` enumeration. The additional values available include the following:

- **NotOnCanceled and OnlyOnCanceled** The `NotOnCanceled` option specifies that the continuation should run only if the previous action completes and is not canceled. The `OnlyOnCanceled` option specifies that the continuation should run only if the previous action is canceled. The section “Canceling tasks and handling exceptions” later in this chapter describes how to cancel a task.
- **NotOnFaulted and OnlyOnFaulted** The `NotOnFaulted` option indicates that the continuation should run only if the previous action completes and does not throw an unhandled exception. The `OnlyOnFaulted` option causes the continuation to run only if the previous action throws an unhandled exception. The section “Canceling tasks and handling exceptions” provides more information on how to manage exceptions in a task.
- **NotOnRanToCompletion and OnlyOnRanToCompletion** The `NotOnRanToCompletion` option specifies that the continuation should run only if the previous action does not complete successfully; it must either be canceled or throw an exception. `OnlyOnRanToCompletion` causes the continuation to run only if the previous action completes successfully.

The following code example shows how to add a continuation to a task that runs only if the initial action does not throw an unhandled exception:

```
Task task2 = ...
task2.Start();
...
task2.Wait(); // Wait at this point until task2 completes
```

A common requirement of applications that invoke operations in parallel is to synchronize tasks. The `Task` class provides the `Wait` method, which implements a simple task-coordination mechanism. Using this method, you can suspend execution of the current thread until the specified task completes, like this:

```
Task task2 = ... task2.Start(); ... task2.Wait(); // Wait at this point until task2 completes
```

You can wait for a set of tasks by using the static `WaitAll` and `WaitAny` methods of the `Task` class. Both methods take a `params` array containing a set of `Task` objects. The `WaitAll` method waits until all specified tasks have completed, and `WaitAny` stops until at least one of the specified tasks has finished. You use them like this:

```
Task.WaitAll(task, task2); // Wait for both task and task2 to complete
Task.WaitAny(task, task2); // Wait for either task or task2 to complete
```

Using the Task class to implement parallelism

In the next exercise, you'll use the Task class to parallelize processor-intensive code in an application, and you'll see how this parallelization reduces the time taken for the application to run by spreading the computations across multiple processor cores. The application, called GraphDemo, consists of a page that uses an Image control to display a graph. The application plots the points for the graph by performing a complex calculation.



Note The exercises in this chapter are intended to be run on a computer with a multicore processor. If you have a single-core CPU, you won't observe the same effects. Additionally, you will run the app in debug mode. This mode is not optimized for maximum performance, but it provides access to performance information that's not available in release mode. As long as you run all the tests performed using this app in the same mode, the results will provide a valid performance comparison. You shouldn't start any additional programs or services between exercises because these might affect the results that you see.

To examine and run the GraphDemo single-threaded application

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **GraphDemo** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 23\GraphDemo** folder in your **Documents** folder. This is a UWP app.
3. In Solution Explorer, in the GraphDemo project, double-click the file MainPage.xaml to display the form in the Design View window.

Apart from the Grid control defining the layout, the form contains the following important controls:

- **An Image control called graphImage** This image control displays the graph rendered by the application.
- **A Button control called plotButton** The user selects this button to generate the data for the graph and display it in the graphImage control.



Note In the interest of keeping the operation of the application in this exercise simple, it displays the button on the page. In a production UWP app, buttons such as this should be located on a command bar.

- **A TextBlock control called duration** The application displays the time taken to generate and render the data for the graph in this label.

4. In Solution Explorer, expand the **MainPage.xaml** file and then double-click **MainPage.xaml.cs** to display the code for the form in the Code and Text Editor window.

The form uses a `WriteableBitmap` object (defined in the `Windows.UI.Xaml.Media.Imaging` namespace) called `graphBitmap` to render the graph. The code in the `plotButton_Click` method creates this object, but the class instance variables `pixelWidth` and `pixelHeight` specify the horizontal and vertical resolution, respectively, for the `WriteableBitmap` object:

```
public partial class MainPage : Window
{
    // Reduce pixelWidth and pixelHeight if there is insufficient memory available
    private int pixelWidth = 10000;
    private int pixelHeight = 5000;
    ...
}
```



Note This application was developed and tested on a desktop computer with 8 GB of memory. (It was also tested on a 4 GB machine.) If your computer has less memory than this available, you might need to reduce the values in the `pixelWidth` and `pixelHeight` variables; otherwise, the application might generate `OutOfMemoryException` exceptions, causing the application to terminate without warning.

Don't try to increase these values if you have a bigger machine; the UWP model has a limit on the amount of memory that an application can use—currently around 2 GB, even on a desktop computer. If you exceed this value, your application might be terminated without warning. The rationale behind this limitation is that many devices on which UWP applications run are memory constrained, and a single app should not be allowed to consume all the memory resources available to the detriment of other apps.

5. Examine the code for the `plotButton_Click` method:

```
private void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
    Random rand = new Random();
    redValue = (byte)rand.Next(0xFF);
    greenValue = (byte)rand.Next(0xFF);
    blueValue = (byte)rand.Next(0xFF);

    int dataSize = bytesPerPixel * pixelWidth * pixelHeight;
    byte data[] = new byte[dataSize];

    Stopwatch watch = Stopwatch.StartNew();
    generateGraphData(data);
    duration.Text = $"Duration (ms): {watch.ElapsedMilliseconds}";
}
```

```

        WriteableBitmap graphBitmap = new WriteableBitmap(pixelWidth, pixelHeight);
        using (Stream pixelStream = graphBitmap.PixelBuffer.AsStream())
        {
            pixelStream.Seek(0, SeekOrigin.Begin);
            pixelStream.Write(data, 0, data.Length);
            graphBitmap.Invalidate();
            graphImage.Source = graphBitmap;
        }
        ...
    }

```

This method runs when the user selects the `plotButton` button. You will select this button several times later in the exercise, which will let you see that a new version of the graph has been drawn each time this method generates a random set of values for the intensity of the red, green, and blue points that are plotted. (The graph will be a different color each time you select this button.)

The next two lines instantiate a byte array that will hold the data for the graph. The size of this array depends on the resolution of the `WriteableBitmap` object, determined by the `pixelWidth` and `pixelHeight` fields. This size must be scaled by the amount of memory required to render each pixel. The `WriteableBitmap` class uses 4 bytes for each pixel, which specify each pixel's relative red, green, and blue intensity and alpha blending value. (The alpha blending value determines the transparency and brightness of the pixel.)

The `watch` variable is a `System.Diagnostics.Stopwatch` object. The `StopWatch` type is useful for timing operations. The static `StartNew` method of the `StopWatch` type creates a new instance of a `StopWatch` object and starts it running. You can query the running time of a `StopWatch` object by examining the `ElapsedMilliseconds` property.

The `generateGraphData` method populates the data array with the data for the graph to be displayed by the `WriteableBitmap` object. You'll examine this method in the next step.

When the `generateGraphData` method has completed, the elapsed time (in milliseconds) appears in the `duration` `TextBox` control.

The final block of code creates the `graphBitmap WriteableBitmap` object. The information held in the data array is copied into this object for rendering. The simplest technique is to create an in-memory stream that can be used to populate the `PixelBuffer` property of the `WriteableBitmap` object. You can then use the `Write` method of this stream to copy the contents of the data array into this buffer. The `Invalidate` method of the `WriteableBitmap` class requests that the operating system redraws the bitmap by using the information held in the buffer. The `Source` property of an `Image` control specifies the data that the `Image` control should display. The final statement sets the `Source` property to the `WriteableBitmap` object.

6. Examine the code for the `generateGraphData` method, shown here:

```
private void generateGraphData(byte[] data)
{
    double a = pixelWidth / 2;
    double b = a * a;
    double c = pixelHeight / 2;
    for (double x = 0; x < a; x++)
    {
        double s = x * x;
        double p = Math.Sqrt(b - s);
        for (double i = -p; i < p; i += 3)
        {
            double r = Math.Sqrt(s + i * i) / a;
            double q = (r - 1) * Math.Sin(24 * r);
            double y = i / 3 + (q * c);
            plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
            plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
        }
    }
}
```

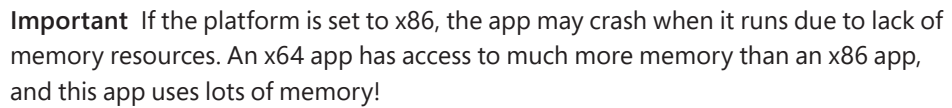
This method performs a series of calculations to plot the points for a rather complex graph. (The actual calculation is unimportant; it just generates a graph that looks attractive.) As it calculates each point, it calls the `plotXY` method to set the appropriate bytes in the data array that correspond to these points. The points for the graph are reflected around the x-axis, so the `plotXY` method is called twice for each calculation: once for the positive value of the x-coordinate, and once for the negative value.

7. Examine the `plotXY` method:

```
private void plotXY(byte[] data, int x, int y)
{
    int pixelIndex = (x + y * pixelWidth) * bytesPerPixel;
    data[pixelIndex] = blueValue;
    data[pixelIndex + 1] = greenValue;
    data[pixelIndex + 2] = redValue;
    data[pixelIndex + 3] = 0xBF;
}
```

This method sets the appropriate bytes in the data array that corresponds to x- and y-coordinates passed in as parameters. Each point plotted corresponds to a pixel, and each pixel consists of 4 bytes, as described earlier. Any pixels left unset are displayed as black. The value `0xBF` for the alpha blend byte indicates that the corresponding pixel should be displayed with a moderate intensity. If you decrease this value, the pixel will become fainter. Setting the value to `0xFF` (the maximum value for a byte) will display the pixel at its brightest intensity.

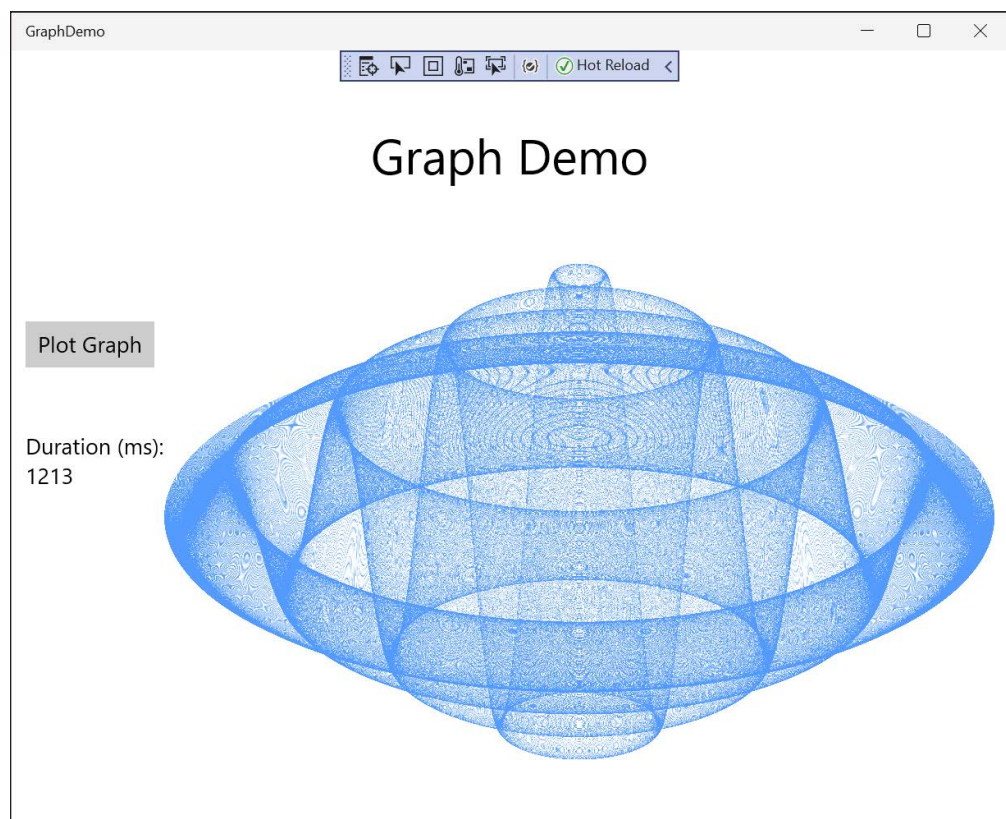
8. In the Visual Studio toolbar, make sure the platform is set to **x64**. Then, on the **Debug** menu, select **Start Debugging** to build and run the application.



- Please be patient. The application typically takes a couple of seconds to generate and display the graph, and the application is unresponsive while this occurs. (Chapter 24 explains why this is so, and also how you can avoid this behavior.)



Note The application was run on a computer with a multicore processor running at 3.10 GHz. Your times might vary if you're using a slower or faster processor with a different number of cores.



10. Select **Plot Graph** again and note the time required to redraw the graph. Repeat this action several times to obtain an average value.



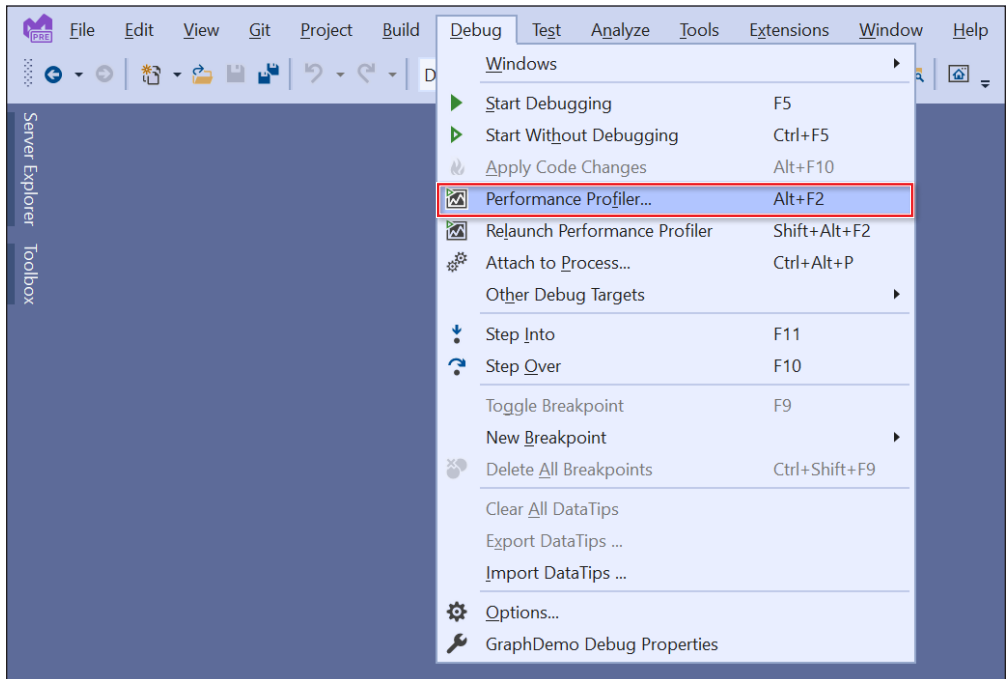
Note You might occasionally find that the graph takes longer to appear. This tends to occur if you're running close to your computer's memory capacity and Windows has to page data between memory and disk. If you encounter this problem, discard this time and don't include it when calculating your average.

11. Return to Visual Studio and stop debugging.

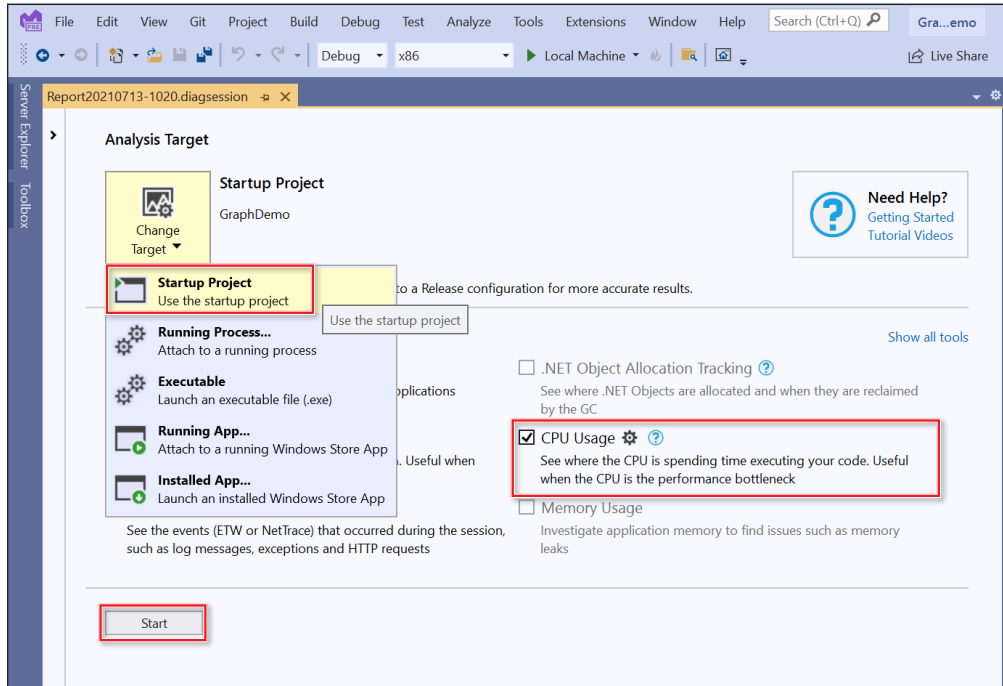
You now have a baseline figure for the performance of the application. In the next exercise, you'll use the Performance Profiler tool in Visual Studio to examine CPU usage when the app runs.

To run the app using Performance Profiler

1. In Visual Studio, on the **Debug** menu, select **Performance Profiler**:



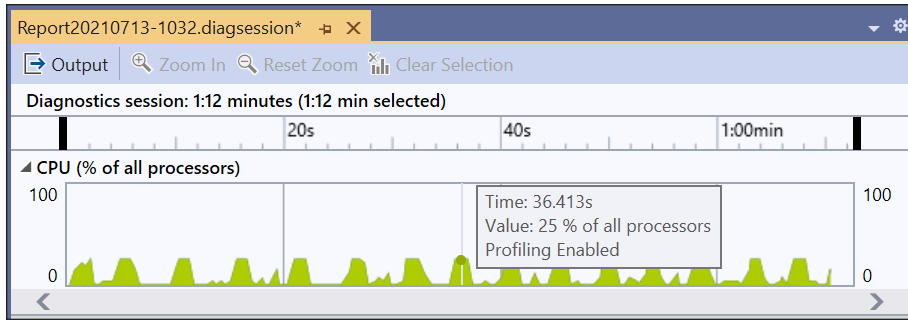
2. In the Analysis Target window, select **Change Target**, and select **Startup Project**.
3. In the **Available Tools** section of the window, select **CPU Usage**. Then select **Start** to run the application using Performance Profiler.



4. When the Graph Demo application starts, select **Plot Graph**.
5. Wait for a few seconds, and then select **Plot Graph** again.
6. Repeat step 4 several times, waiting a few seconds between selections.
7. Close the application and return to Visual Studio.

Performance Profiler generates a report showing how the CPU was used. The graph at the top of the report should show several flat-topped peaks. These peaks will correspond to each time you selected Plot Graph.

8. Hover over one of the peaks to view the CPU use at that point. In the following image, the maximum CPU use is around 25%. In my case, the application was running on a quad-core machine, but it looks as though the app is only using a single processor:



9. Close the report without saving it.

It's clear from the CPU use displayed by Performance Profiler that the application is not making full use of the processing resources available. On a dual-core machine, it's using just over half of the CPU power, and on a quad-core machine, it's employing a little more than a quarter of the CPU. This is because the application is single-threaded, and in a Windows application, a single thread can use only a single core on a multicore processor. To spread the load over all the available cores, you must divide the application into tasks and arrange for each task to be executed by a separate thread—each on a different core. This is what you'll do next.

To modify the GraphDemo application to use Task objects

1. Return to Visual Studio 2022 and display the MainPage.xaml.cs file in the Code and Text Editor window if it is not already open.
2. Examine the generateGraphData method. The purpose of this method is to populate the items in the data array. It iterates through the array by using the outer for loop based on the **x** loop control variable, highlighted in bold here:

```
private void generateGraphData(byte[] data)
{
    double a = pixelWidth / 2;
    double b = a * a;
    double c = pixelHeight / 2;
    for (double x = 0; x < a; x ++)
    {
        double s = x * x;
        double p = Math.Sqrt(b - s);
        for (double i = -p; i < p; i += 3)
        {
            double r = Math.Sqrt(s + i * i) / a;
            double q = (r - 1) * Math.Sin(24 * r);
            double y = i / 3 + (q * c);
            plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
            plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
        }
    }
}
```

The calculation performed by one iteration of this loop is independent of the calculations performed by the other iterations. Therefore, it makes sense to partition the work performed by this loop and run different iterations on a separate processor.

3. Modify the definition of the `generateGraphData` method to take two additional `int` parameters, called `partitionStart` and `partitionEnd`, as shown in bold in the following example:

```
private void generateGraphData(byte[] data, int partitionStart, int partitionEnd)
{
    ...
}
```

4. In the `generateGraphData` method, change the outer `for` loop to iterate between the values of `partitionStart` and `partitionEnd`, as shown here in bold:

```
private void generateGraphData(byte[] data, int partitionStart, int partitionEnd)
{
    ...
    for (int x = partitionStart; x < partitionEnd; x++)
    {
        ...
    }
}
```

5. In the Code and Text Editor window, add the following `using` directive to the list at the top of the `MainPage.xaml.cs` file:

```
using System.Threading.Tasks;
```

6. In the `plotButton_Click` method, comment out the statement that calls the `generateGraphData` method and add the statement shown in bold in the following code. This new statement creates a `Task` object and starts it running:

```
...
Stopwatch watch = Stopwatch.StartNew();
// generateGraphData(data);
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4));
...
```

The task runs the code specified by the lambda expression. The values for the `partitionStart` and `partitionEnd` parameters indicate that the `Task` object calculates the data for the first half of the graph. (The data for the complete graph consists of points plotted for the values between 0 and `pixelWidth / 2`.)

7. Add another statement that creates and runs a second `Task` object on another thread, as shown in the following bold code:

```
...
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4));
Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4, pixelWidth / 2));
...
```

This `Task` object invokes the `generateGraphData` method and calculates the data for the values between `pixelWidth / 4` and `pixelWidth / 2`.

8. Add the following statement shown in bold that waits for both Task objects to complete their work before continuing:

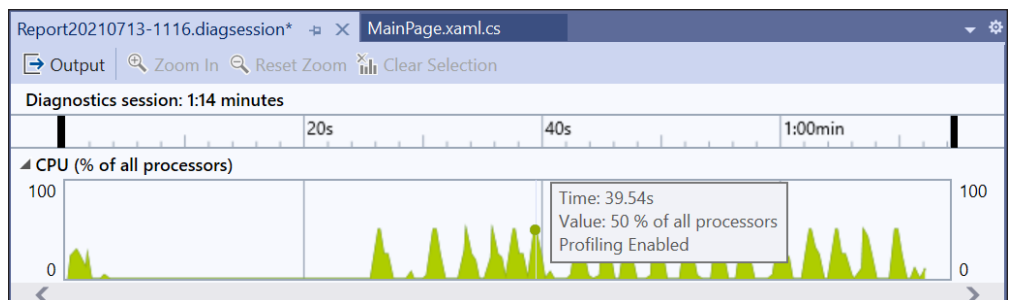
```
...  
Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4, pixelWidth / 2));  
Task.WaitAll(first, second);  
...
```

9. On the **Debug** menu, select **Start Without Debugging** to build and run the application.
10. In the Graph Demo window, select **Plot Graph** and note the duration of the operation.
11. Repeat this step several times to get an average value. You should see that the application runs significantly quicker than it did previously. On my computer, the typical time dropped to 654 milliseconds—a reduction in time of approximately 45 percent.

In most cases, the time required to perform the calculations will be cut by nearly half. However, the application still has some single-threaded elements, such as the logic that actually displays the graph after the data has been generated. This is why the overall time is still more than half the time taken by the previous version of the application.

12. Stop the application and return to Visual Studio.
13. On the **Debug** menu, select **Relaunch Performance Profiler**. Performance Profiler starts using the performance settings you specified earlier.
14. Select **Plot Graph** several times, waiting for a few seconds between selections. Note that the performance will have dropped slightly due to the effects of Performance Profiler.
15. Close the app and wait for Performance Profiler to generate the report.

In the CPU use graph, you should see that the application uses more CPU resources than before. On my quad-core machine, the CPU use peaked at approximately 50 percent each time I selected Plot Graph. This happens because the two tasks were each run on separate cores, but the remaining two cores were left unoccupied. If you have a dual-core machine, you will likely see processor use briefly exceed 90 percent each time the graph is generated.





Note If you have a quad-core computer, you can increase the CPU use and further reduce the time by adding two more Task objects and dividing the work into four chunks in the `plotButton_Click` method, as shown here in bold:

```
...
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 8));
Task second = Task.Run(() => generateGraphData(data, pixelWidth / 8,
pixelWidth / 4));
Task third = Task.Run(() => generateGraphData(data, pixelWidth / 4,
pixelWidth * 3 / 8));
Task fourth = Task.Run(() => generateGraphData(data, pixelWidth * 3 / 8,
pixelWidth / 2));
Task.WaitAll(first, second, third, fourth);
...
```

If you have only a dual-core processor, you can still try this modification, and you should notice a small beneficial effect on the time. This is primarily because of how the algorithms used by the CLR optimize the way in which the threads for each task are scheduled.

Abstracting tasks by using the Parallel class

Using the Task class gives you complete control over the number of tasks your application creates. However, you had to modify the design of the application to accommodate the use of Task objects. You also had to add code to synchronize operations; the application can render the graph only when all the tasks have completed. In a complex application, the synchronization of tasks can become a non-trivial process that is easily prone to mistakes.

With the `Parallel` class, you can parallelize some common programming constructs without having to redesign an application. Internally, the `Parallel` class creates its own set of Task objects, and it synchronizes these tasks automatically when they've completed.

The `Parallel` class is located in the `System.Threading.Tasks` namespace and provides a small set of static methods that you can use to indicate that code should be run in parallel if possible. These methods are as follows:

- **Parallel.For** You can use this method in place of a C# `for` statement. It defines a loop in which iterations can run in parallel by using tasks. This method is heavily overloaded, but the general principle is the same for each overload: you specify a start value, an end value, and a reference to a method that takes an integer parameter. The method is executed for every value between the start value and one below the end value specified, and the parameter is populated with an integer that specifies the current value. For example, consider the following simple `for` loop that performs each iteration in sequence:

```
for (int x = 0; x < 100; x++)
{
    // Perform loop processing
}
```

Depending on the processing performed by the body of the loop, you might be able to replace this loop with a `Parallel.For` construct that can perform iterations in parallel, like this:

```
Parallel.For(0, 100, performLoopProcessing);
...
private void performLoopProcessing(int x)
{
    // Perform loop processing
}
```

Using the overloads of the `Parallel.For` method, you can provide local data that's private to each thread, specify various options for creating the tasks run by the `For` method, and create a `ParallelLoopState` object that can be used to pass state information to other concurrent iterations of the loop. (Using a `ParallelLoopState` object is described later in this chapter, in the section "Canceling tasks and handling exceptions.")

- **`Parallel.ForEach<T>`** You can use this method in place of a C# `foreach` statement. Like the `For` method, `ForEach` defines a loop in which iterations can run in parallel. You specify a collection that implements the `IEnumerable<T>` generic interface and a reference to a method that takes a single parameter of type `T`. The method is executed for each item in the collection, and the item is passed as the parameter to the method. Overloads are available with which you can provide private local thread data and specify options for creating the tasks run by the `ForEach` method.
- **`Parallel.Invoke`** You can use this method to execute a set of parameter-less method calls as parallel tasks. You specify a list of delegated method calls (or lambda expressions) that take no parameters and don't return values. Each method call can be run on a separate thread, in any order. For example, the following code makes a series of method calls:

```
doWork();
doMoreWork();
doYetMoreWork();
```

You can replace these statements with the following code, which invokes these methods by using a series of tasks:

```
Parallel.Invoke(
    doWork,
    doMoreWork,
    doYetMoreWork
);
```

The `Parallel` class determines the degree of parallelism appropriate for the environment and workload of the computer. For example, if you use `Parallel.For` to implement a loop that performs 1,000 iterations, the `Parallel` class doesn't necessarily create 1,000 concurrent tasks (unless you have an exceptionally powerful processor with 1,000 cores). Instead, the `Parallel` class creates what it considers to be the optimal number of tasks that balances the available resources against the requirement to keep the processors occupied. A single task might perform multiple iterations, and the tasks will coordinate with each other to determine which iterations each task will perform. An important consequence of this is that you cannot guarantee the order in which the iterations are executed, so

you must ensure that there are no dependencies between iterations; otherwise, you might encounter unexpected results, as you'll see later in this chapter.

In the next exercise, you'll return to the original version of the GraphDemo application and use the `Parallel` class to perform operations concurrently.

To use the `Parallel` class to parallelize operations in the GraphDemo application

1. Using Visual Studio 2022, open the **GraphDemo** solution, which is located in the `\Microsoft Press\VCBS\Chapter 23\Parallel GraphDemo` folder in your **Documents** folder.

This is a copy of the original GraphDemo application. It does not use tasks yet.

2. In Solution Explorer, in the GraphDemo project, expand the **MainPage.xaml** node, and then double-click **MainPage.xaml.cs** to display the code for the form in the Code and Text Editor window.

3. Add the following `using` directive to the list at the top of the file:

```
using System.Threading.Tasks;
```

4. Locate the `generateGraphData` method. It looks like this:

```
private void generateGraphData(byte[] data)
{
    double a = pixelWidth / 2;
    double b = a * a;
    double c = pixelHeight / 2;
    for (double x = 0; x < a; x++)
    {
        double s = x * x;
        double p = Math.Sqrt(b - s);
        for (double i = -p; i < p; i += 3)
        {
            double r = Math.Sqrt(s + i * i) / a;
            double q = (r - 1) * Math.Sin(24 * r);
            double y = i / 3 + (q * c);
            plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
            plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
        }
    }
}
```

The outer `for` loop that iterates through values of the variable `x` is a prime candidate for parallelization. You might also consider the inner loop based on the variable `i`. However, if you have nested loops such as those that occur in this code, it's good practice to parallelize the outer loops first and then test to see whether the performance of the application is sufficient. If it's not, work your way through nested loops and parallelize them starting with outer loops and moving to inner loops, testing the performance after you modify each one. You'll find that in many cases, parallelizing outer loops has the most impact on performance, whereas the effects of modifying inner loops are more marginal.

5. Cut the local variable definitions and the code in the body of the for loop and create a new private void method called `calculateData` with this code. The `calculateData` method should take an `int` parameter called `x` and a byte array called `data`. Also, move the statements that declare the local variables `a`, `b`, and `c` from the `generateGraphData` method to the start of the `calculateData` method. The following code shows the `generateGraphData` method with this code removed and the `calculateData` method added (do not try to compile this code yet):

```
private void generateGraphData(byte[] data)
{
    for (double x = 0; x < a; x++)
    {
    }
}

private void calculateData(int x, byte[] data)
{
    double a = pixelWidth / 2;
    double b = a * a;
    double c = pixelHeight / 2;
    double s = x * x;
    double p = Math.Sqrt(b - s);
    for (double i = -p; i < p; i += 3)
    {
        double r = Math.Sqrt(s + i * i) / a;
        double q = (r - 1) * Math.Sin(24 * r);
        double y = i / 3 + (q * c);
        plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
        plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
    }
}
```

6. In the `generateGraphData` method, replace the for loop with the following statement shown in bold that calls the static `Parallel.For` method:

```
private void generateGraphData(byte[] data)
{
    Parallel.For(0, pixelWidth / 2, x => calculateData(x, data));
}
```

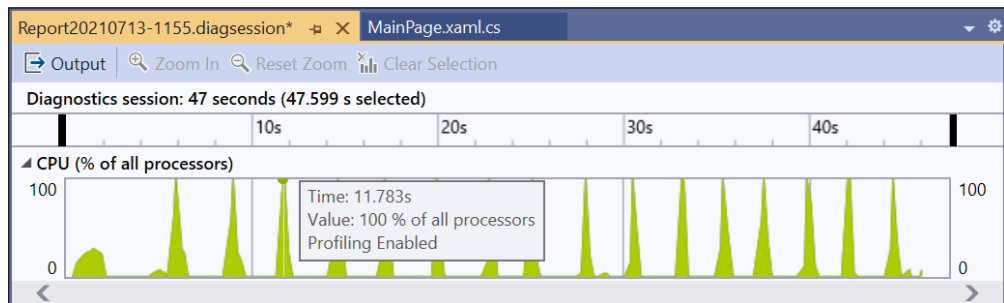
This code is the parallel equivalent of the original for loop. It iterates through the values from 0 to `pixelWidth / 2 - 1` inclusive. Each invocation runs by using a task, and each task might run more than one iteration. The `Parallel.For` method finishes only when all the tasks it has created complete their work. Remember: the `Parallel.For` method expects the final parameter to be a method that takes a single integer parameter. It calls this method passing the current loop index as the parameter. In this example, the `calculateData` method doesn't match the required signature because it takes two parameters: an integer and a byte array. For this reason, the code uses a lambda expression that acts as an adapter that calls the `calculateData` method with the appropriate arguments.

7. On the **Debug** menu, select **Start Debugging** to build and run the application.
8. In the Graph Demo window, select **Plot Graph**. When the graph appears in the Graph Demo window, record the time taken to generate the graph. Repeat this action several times to get an average value.

You should notice that the application runs as least as quickly as the previous version that used Task objects and possibly faster, depending on the number of CPUs you have available.

Close the GraphDemo application and return to Visual Studio.

9. On the **Debug** menu, select **Relaunch Performance Profiler**. The app starts again, using Performance Profiler to track CPU use.
10. Select **Plot Graph** several times, waiting for a few seconds between selections.
11. Close the app and return to Visual Studio.
12. In the CPU usage graph, examine the CPU use. It should peak at 100 percent each time Plot Graph was selected. This is because the `Parallel.For` construct automatically takes advantage of all the available processors. So, on a dual-core machine, it will use two processor cores; on a quad-core machine, it will use four processor cores; and so on. You don't have to amend your code in any way to scale between processor architectures.



When not to use the `Parallel` class

Despite appearances and the best efforts of the Microsoft .NET development team, the `Parallel` class is not magic. You cannot use it without due consideration and just expect your applications to suddenly run significantly faster and produce the same results. The purpose of the `Parallel` class is to parallelize CPU-bound independent areas of your code.

If you're not running CPU-bound code, parallelizing it might not improve performance. In this case, the overhead of creating a task, running this task on a separate thread, and waiting for the task to complete is likely to be greater than the cost of running this method directly. The additional overhead might account for only a few milliseconds each time a method is called, but you should bear in mind the number of times a method runs. If the method call is located in a nested loop and is executed thousands of times, all these small overhead costs will add up. The general rule is to use `Parallel.Invoke` only when it is worthwhile. You should reserve `Parallel.Invoke` for operations that are computationally intensive; otherwise, the overhead of creating and managing tasks can actually slow down an application.

The other key consideration for using the `Parallel` class is that parallel operations should be independent. For example, if you attempt to use `Parallel.For` to parallelize a loop in which iterations have a dependency on each other, the results will be unpredictable. To see what I mean, look

at the following code. (You can find this example in the `ParallelLoop` solution, which is located in the `\Microsoft Press\VCSBS\Chapter 23\ParallelLoop` folder in your Documents folder.)

```
using System;
using System.Threading;
using System.Threading.Tasks;
namespace ParallelLoop
{
    class Program
    {
        private static int accumulator = 0;

        static void Main(string[] args)
        {
            for (int i = 0; i < 100; i++)
            {
                AddToAccumulator(i);
            }
            Console.WriteLine($"Accumulator is {accumulator}");
        }

        private static void AddToAccumulator(int data)
        {
            if ((accumulator % 2) == 0)
            {
                accumulator += data;
            }
            else
            {
                accumulator -= data;
            }
        }
    }
}
```

This program iterates through values from 0 to 99 and calls the `AddToAccumulator` method with each value in turn. The `AddToAccumulator` method examines the current value of the `accumulator` variable. If it is even, it adds the value of the parameter to the `accumulator` variable; otherwise, it subtracts the value of the parameter. At the end of the program, the result is displayed. If you run this program, the value output should be `-100`.

To increase the degree of parallelism in this simple application, you might be tempted to replace the `for` loop in the `Main` method with `Parallel.For`. For, as shown here in bold:

```
static void Main(string[] args)
{
    Parallel.For (0, 100, AddToAccumulator);
    Console.WriteLine($"Accumulator is {accumulator}");
}
```

However, there's no guarantee that the tasks created to run the various invocations of the `AddToAccumulator` method will execute in any specific sequence. (The code is also not thread-safe because multiple threads running the tasks might attempt to modify the `accumulator` variable concurrently.) The value calculated by the `AddToAccumulator` method depends on the sequence being maintained, so the result of this modification is that the application might now generate different

values each time it runs. In this simple case, you might not actually see any difference in the value calculated because the `AddToAccumulator` method runs very quickly and .NET might elect to run each invocation sequentially by using the same thread. However, if you make the following change shown in bold to the `AddToAccumulator` method, you will get different results:

```
private static void AddToAccumulator(int data)
{
    if ((accumulator % 2) == 0)
    {
        accumulator += data;
        Thread.Sleep(10); // wait for 10 milliseconds
    }
    else
    {
        accumulator -= data;
    }
}
```

The `Thread.Sleep` method simply causes the current thread to wait for the specified period of time. This modification simulates performing additional processing on a thread. It also affects how the `Parallel` class schedules the tasks, which now run on different threads, resulting in a different sequence that calculates a different value. If you run the app several times, you'll get a different result each time.

The general rule is to use `Parallel.For` and `Parallel.ForEach` only if you can guarantee that each iteration of the loop is independent and to test your code thoroughly. A similar consideration applies to `Parallel.Invoke`: use this construct to make method calls only if they're independent and the application doesn't depend on them being run in a particular sequence.

Canceling tasks and handling exceptions

A common requirement of applications that perform long-running operations is the ability to stop those operations if necessary. However, you shouldn't simply abort a task, as this could leave the data in your application in an indeterminate state. Instead, the `Task` class implements a cooperative cancellation strategy. *Cooperative cancellation* enables a task to select a convenient point at which to stop processing and enables it to undo any work it has performed prior to cancellation if necessary.

The mechanics of cooperative cancellation

Cooperative cancellation is based on the notion of a cancellation token. A *cancellation token* is a structure that represents a request to cancel one or more tasks.

The method a task runs should include a `System.Threading.CancellationToken` parameter. An application that wants to cancel the task sets the Boolean `IsCancellationRequested` property of this parameter to `true`. The method running in the task can query this property at various points during its processing. If this property is set to `true` at any point, it knows that the application has requested that the task be canceled. Also, the method knows what work it has done so far, so it can undo any changes if necessary and then finish. Alternatively, the method can simply ignore the request and continue running.



Tip You should frequently examine a task's cancellation token, but not so frequently that you adversely affect the performance of the task. If possible, you should aim to check for cancellation at least every 10 milliseconds, but no more frequently than every millisecond.

An application obtains a `CancellationToken` by creating a `System.Threading.CancellationTokenSource` object and querying its `Token` property. The application can then pass this `CancellationToken` object as a parameter to any methods started by tasks that the application creates and runs. If the application needs to cancel the tasks, it calls the `Cancel` method of the `CancellationTokenSource` object. This method sets the `IsCancellationRequested` property of the `CancellationToken` passed to all the tasks.

The code example that follows shows how to create a cancellation token and use it to cancel a task. The `initiateTasks` method instantiates the `cancellationTokenSource` variable and obtains a reference to the `CancellationToken` object available through this variable. The code then creates and runs a task that executes the `doWork` method. Later, the code calls the `Cancel` method of the cancellation token source, which sets the cancellation token. The `doWork` method queries the `IsCancellationRequested` property of the cancellation token. If the property is set, the method terminates; otherwise, it continues running.

```
public class MyApplication
{
    ...
    // Method that creates and manages a task
    private void initiateTasks()
    {
        // Create the cancellation token source and obtain a cancellation token
        CancellationTokenSource cancellationTokenSource = new CancellationTokenSource();
        CancellationToken cancellationToken = cancellationTokenSource.Token;
        // Create a task and start it running the doWork method
        Task myTask = Task.Run(() => doWork(cancellationToken));
        ...
        if (...)
        {
            // Cancel the task
            cancellationTokenSource.Cancel();
        }
        ...
    }

    // Method run by the task
    private void doWork(CancellationToken token)
    {
        ...
        // If the application has set the cancellation token, finish processing
        if (token.IsCancellationRequested)
        {
            // Tidy up and finish
            ...
            return;
        }
    }
}
```

```

        // If the task hasn't been canceled, continue running as normal
        ...
    }
}

```

In addition to providing a high degree of control over the cancellation processing, this approach is scalable across any number of tasks. You can start multiple tasks and pass the same `CancellationToken` object to each of them. If you call `Cancel` on the `CancellationTokenSource` object, each task will check whether the `IsCancellationRequested` property has been set and proceed accordingly.

You can also register a callback method (in the form of an `Action` delegate) with the cancellation token by using the `Register` method. When an application invokes the `Cancel` method of the corresponding `CancellationTokenSource` object, this callback runs. However, you cannot guarantee when this method executes. It might be before or after the tasks have performed their own cancellation processing, or even during that process.

```

...
cancellationToken.Register(doAdditionalWork);
...
private void doAdditionalWork()
{
    // Perform additional cancellation processing
}

```

In the next exercise, you'll add cancellation functionality to the `GraphDemo` application.

To add cancellation functionality to the `GraphDemo` application

1. Using Visual Studio 2022, open the **GraphDemo** solution, which is located in the `\Microsoft Press\VCSBS\Chapter 23\GraphDemo With Cancellation` folder in your **Documents** folder.

This is a completed copy of the `GraphDemo` application from the earlier exercise that uses tasks to improve processing throughput. The user interface also includes a `Cancel` button that the user can select to stop the tasks that calculate the data for the graph. This button runs the `cancelButton_Click` method. Additionally, the size of the `pixelWidth` and `pixelHeight` variables have been increased to plot the graph in a higher resolution and extend the time taken to generate the graph data. (You might need to reduce the values of these variables if you have less than 8 GB of memory available on your computer.)



Important Because of the increased memory requirements, be sure you have selected the x64 platform in the Visual Studio toolbar before you attempt to run this application.

2. In Solution Explorer, in the GraphDemo project, double-click **MainPage.xaml** to display the form in the Design View window. Note the Cancel button that appears in the left pane of the form.
3. Open the MainPage.xaml.cs file in the Code and Text Editor window and locate the `CancelButton_Click` method. This method runs when the user selects Cancel. It is currently empty.
4. Add the following `using` directive to the list at the top of the file:

```
using System.Threading;
```

The types used by cooperative cancellation reside in this namespace.

5. Add a `CancellationTokenSource` field called `tokenSource` to the `MainPage` class and initialize it to `null`, as shown in the following code in bold:

```
public sealed partial class MainPage : Page
{
    ...
    private byte redValue, greenValue, blueValue;
    private CancellationTokenSource tokenSource = null;
    ...
}
```

6. Find the `generateGraphData` method and add a `CancellationToken` parameter called `token` to the method definition, as shown here in bold:

```
private void generateGraphData(byte[] data, int partitionStart, int partitionEnd,
CancellationToken token)
{
    ...
}
```

7. In the `generateGraphData` method, at the start of the inner `for` loop, add the following code shown in bold to check whether cancellation has been requested. If so, return from the method; otherwise, continue calculating values and plotting the graph.

```
private void generateGraphData(byte[] data, int partitionStart, int partitionEnd,
CancellationToken token)
{
    double a = pixelWidth / 2;
    double b = a * a;
    double c = pixelHeight / 2;
    for (double x = partitionStart; x < partitionEnd; x++)
    {
        double s = x * x;
        double p = Math.Sqrt(b - s);
        for (double i = -p; i < p; i += 3)
        {
            if (token.IsCancellationRequested)
            {
                return;
            }
        }
    }
}
```

```

        double r = Math.Sqrt(s + i * i) / a;
        double q = (r - 1) * Math.Sin(24 * r);
        double y = i / 3 + (q * c);
        plotXY(data, (int)(-x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
        plotXY(data, (int)(x + (pixelWidth / 2)), (int)(y + (pixelHeight / 2)));
    }
}
}

```

8. In the `plotButton_Click` method, add the following statements shown in bold that instantiate the `tokenSource` variable and retrieve the `CancellationToken` object into a variable called `token`:

```

private void plotButton_Click(object sender, RoutedEventArgs e)
{
    Random rand = new Random();
    redValue = (byte)rand.Next(0xFF);
    greenValue = (byte)rand.Next(0xFF);
    blueValue = (byte)rand.Next(0xFF);

    tokenSource = new CancellationTokenSource();
    CancellationToken token = tokenSource.Token;
    ...
}

```

9. Modify the statements in the `plotButton_Click` method that create and run the two tasks and pass the `token` variable as the final parameter to the `generateGraphData` method as shown in bold:

```

...
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4, token));
Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4,
pixelWidth / 2, token));
...

```

10. Edit the definition of the `plotButton_Click` method and add the `async` modifier, as shown here in bold:

```

private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
}

```

11. In the body of the `plotButton_Click` method, comment out the `Task.WaitAll` statement that waits for the tasks to complete and replace it with the following statements in bold that use the `await` operator instead.

```

...
// Task.WaitAll(first, second);
await first;
await second;

duration.Text = ...;
...

```

The changes in these two steps are necessary because of the single-threaded nature of the Windows user interface. Under normal circumstances, when an event handler for a user interface component such as a button starts running, event handlers for other user interface components are blocked until the first event handler completes (even if the event handler is using tasks). In this example, using the `Task.WaitAll` method to wait for the tasks to complete would render the Cancel button useless because the event handler for the Cancel button will not run until the handler for the Plot Graph button completes, in which case there's no point in attempting to cancel the operation. In fact, as mentioned, when you select the Plot Graph button, the user interface is completely unresponsive until the graph appears and the `plotButton_Click` method finishes.

The `await` operator is designed to handle situations such as this. You can use this operator only inside a method marked as `async`. Its purpose is to release the current thread and wait for a task to complete in the background. When that task finishes, control returns to the method, which continues with the next statement. In this example, the two `await` statements simply allow each of the tasks to complete in the background. After the second task has finished, the method continues, displaying the time taken for these tasks to complete in the `duration TextBlock`. Note that it's not an error to await for a task that has already completed; the `await` operator will simply return immediately and pass control to the following statement.



More Info Chapter 24 discusses the `async` modifier and the `await` operator in detail.

12. Find the `cancelButton_Click` method. Then add the code shown here in bold to this method:

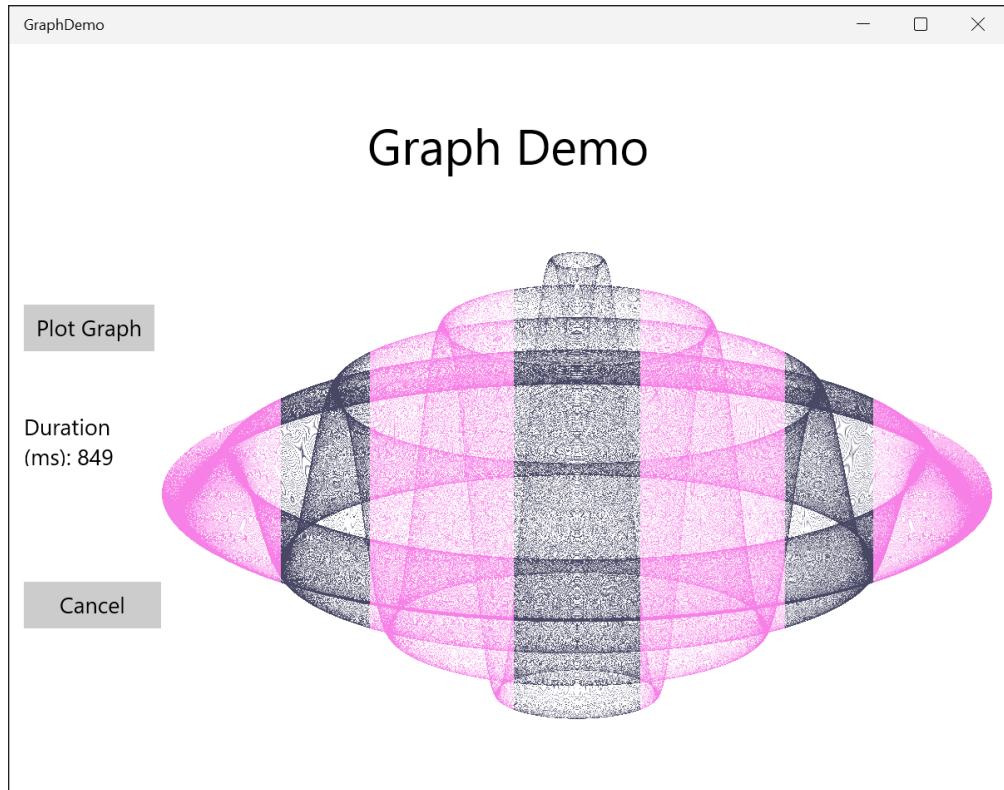
```
private void cancelButton_Click(object sender, RoutedEventArgs e)
{
    if (tokenSource is not null)
    {
        tokenSource.Cancel();
    }
}
```

This code checks that the `tokenSource` variable has been instantiated. If it has, the code invokes the `Cancel` method on this variable.

13. On the **Debug** menu, select **Start Without Debugging** to build and run the application.
14. In the GraphDemo window, select **Plot Graph**, and verify that the graph appears as it did before. Notice that it takes longer to generate the graph. This is because of the increasing number of iterations performed by the `generateGraphData` method.

15. Select **Plot Graph** again, and then quickly select **Cancel**.

If you are swift and select Cancel before the data for the graph is generated, this action causes the methods being run by the tasks to return. The data is not complete, so the graph appears with holes, as shown in the following figure. The size of the holes depends on how quickly you selected Cancel.



16. Close the GraphDemo application and return to Visual Studio.

You can determine whether a task completed or was canceled by examining the `Status` property of the `Task` object. The `Status` property contains a value from the `System.Threading.Tasks.TaskStatus` enumeration. The following list describes some of the status values that you might commonly encounter (there are others):

- **Created** This is the initial state of a task. It has been created but has not yet been scheduled to run.
- **WaitingToRun** The task has been scheduled but has not yet started to run.
- **Running** The task is currently being executed by a thread.
- **RanToCompletion** The task completed successfully without any unhandled exceptions.

- **Canceled** The task was canceled before it could start running, or it acknowledged cancellation and completed without throwing an exception.
- **Faulted** The task terminated because of an exception.

In the next exercise, you'll report the status of each task so that you can see when they have completed or have been canceled.

Canceling a Parallel For or ForEach loop

The `Parallel.For` and `Parallel.ForEach` methods don't provide you with direct access to the `Task` objects that have been created. You don't even know how many tasks are running. The .NET runtime uses its own heuristics to determine the optimal number to use based on the resources available and the current workload of the computer.

If you want to stop the `Parallel.For` or `Parallel.ForEach` method early, you must use a `ParallelLoopState` object. The method you specify as the body of the loop must include an additional `ParallelLoopState` parameter. The `Parallel` class creates a `ParallelLoopState` object and passes it as this parameter into the method. The `Parallel` class uses this object to hold information about each method invocation. The method can call the `Stop` method of this object to indicate that the `Parallel` class should not attempt to perform any iterations beyond those that have already started and finished.

The example that follows shows the `Parallel.For` method calling the `doLoopWork` method for each iteration. The `doLoopWork` method examines the iteration variable. If it is greater than 600, the method calls the `Stop` method of the `ParallelLoopState` parameter. This causes the `Parallel.For` method to stop running further iterations of the loop. (Iterations currently running might continue to completion.)



Note Remember that the iterations in a `Parallel.For` loop are not run in a specific sequence. Consequently, canceling the loop when the iteration variable has the value 600 does not guarantee that the previous 599 iterations have already run. Likewise, some iterations with values greater than 600 might already have completed.

```
Parallel.For(0, 1000, doLoopWork);
...
private void doLoopWork(int i, ParallelLoopState p)
{
    ...
    if (i > 600)
    {
        p.Stop();
    }
}
```

To display the status of each task

1. In Visual Studio, display the MainPage.xaml file in the Design View window.
2. In the XAML pane, add the following markup to the definition of the MainPage form before the final closing `</Grid>` tag, as shown in bold in the following code:

```
        <Image x:Name="graphImage" Grid.Column="1" Stretch="Fill" />
    </Grid>
    <TextBlock x:Name="messages" Grid.Row="4" FontSize="18"
HorizontalAlignment="Left"/>
</Grid>
</Page>
```

This markup adds a `TextBlock` control named `messages` to the bottom of the form.

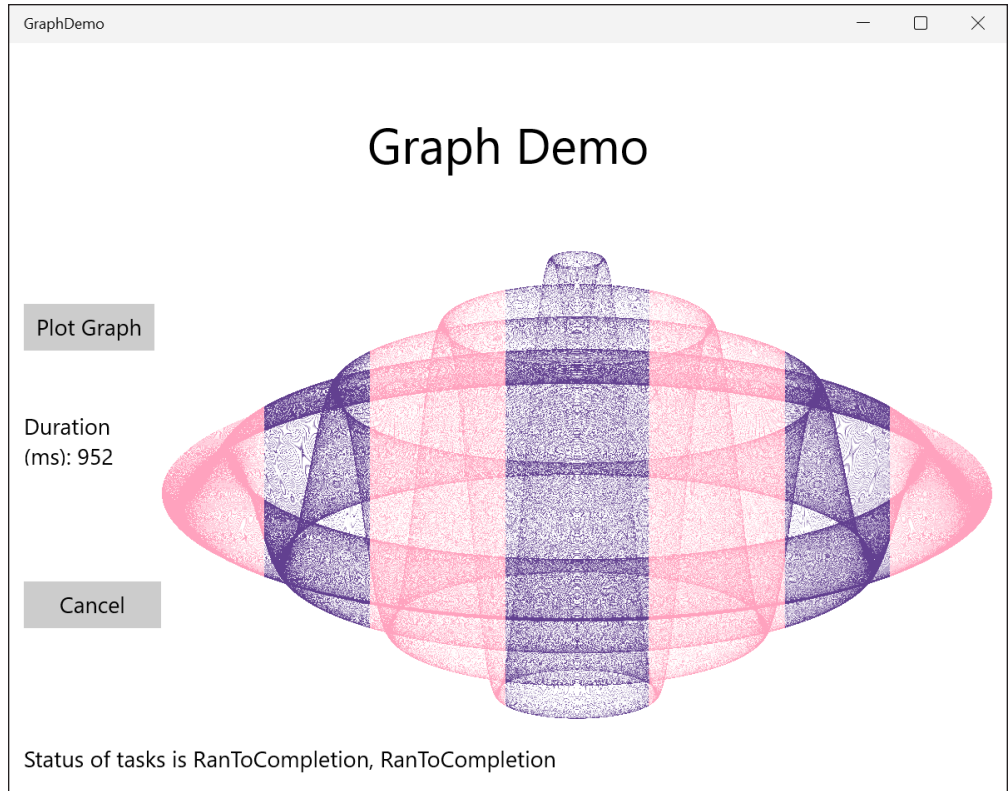
3. Display the MainPage.xaml.cs file in the Code and Text Editor window and find the `plotButton_Click` method.
4. Add the code shown next in bold to this method. These statements generate a string that contains the status of each task after it has finished running and then display this string in the `messages` `TextBlock` control at the bottom of the form.

```
private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
    await first;
    await second;

    duration.Text = $"Duration (ms): {watch.ElapsedMilliseconds}";
    string message = $"Status of tasks is {first.Status}, {second.Status}";
    messages.Text = message;
    ...
}
```

5. On the **Debug** menu, select **Start Without Debugging**.
6. In the GraphDemo window, select **Plot Graph**, but do not select Cancel. Verify that the message displayed reports that the status of the tasks is `RanToCompletion` (two times).
7. In the GraphDemo window, select **Plot Graph** again, and then quickly select **Cancel**.

Surprisingly, the message that appears still reports the status of each task as `RanToCompletion`, even though the graph appears with holes. This occurs because although you sent a cancellation request to each task by using the cancellation token, the methods they were running simply returned. The .NET runtime doesn't know whether the tasks were actually canceled or whether they were allowed to run to completion, and it just ignores the cancellation requests.



8. Close the GraphDemo application and return to Visual Studio.

So, how do you indicate that a task has been canceled rather than allowed to run to completion? The answer lies in the `CancellationToken` object passed as a parameter to the method that the task is running.

The `CancellationToken` class provides a method called `ThrowIfCancellationRequested`. This method tests the `IsCancellationRequested` property of a cancellation token. If it is `true`, the method throws an `OperationCanceledException` exception and aborts the method that the task is running.

The application that started the thread should be prepared to catch and handle this exception, but this leads to another dilemma: if a task terminates by throwing an exception, it actually reverts to the `Faulted` state. This is true even if the exception is an `OperationCanceledException` exception. A task enters the `Canceled` state only if it is canceled without throwing an exception. So, how does a task throw an `OperationCanceledException` without it being treated as an exception?

This time, the answer lies in the task itself. For a task to recognize that an `OperationCanceledException` exception is the result of canceling the task in a controlled manner and not just an exception caused by other circumstances, it must know that the operation has actually been canceled. It can do this only if it can examine the cancellation token. You passed this token as a parameter to the method run by the task, but the task doesn't actually check any of these parameters. Instead, you specify the cancellation token when you create and run the task. The code that follows shows an example based on the `GraphDemo` application. Notice how the token parameter is passed to the `generateGraphData` method (as before) but also as a separate parameter to the `Run` method.

```
tokenSource = new CancellationTokenSource();
CancellationToken token = tokenSource.Token;
...
Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4, token), token);
```

Now, when the method being run by the task throws an `OperationCanceledException` exception, the infrastructure behind the task examines the `CancellationToken`. If it indicates that the task has been canceled, the infrastructure sets the status of the task to `Canceled`. If you're using the `await` operator to wait for the tasks to complete, you also need to be prepared to catch and handle the `OperationCanceledException` exception. This is what you'll do in the next exercise.

To acknowledge cancellation and handle the `OperationCanceledException` exception

1. In Visual Studio, return to the Code and Text Editor window displaying the `MainPage.xaml.cs` file. Then, in the `plotButton_Click` method, modify the statements that create and run the tasks and specify the `CancellationToken` object as the second parameter to the `Run` method (and also as a parameter to the `generateGraphData` method), as shown in bold in the following code:

```
private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
    tokenSource = new CancellationTokenSource();
    CancellationToken token = tokenSource.Token;
    ...
    Task first = Task.Run(() => generateGraphData(data, 0, pixelWidth / 4, token), token);
    Task second = Task.Run(() => generateGraphData(data, pixelWidth / 4,
pixelWidth / 2, token), token);
    ...
}
```

2. Add a `try` block around the statements that run the tasks, wait for them to complete, and display the elapsed time. Then add a `catch` block that handles the `OperationCanceledException` exception. In this exception handler, display the reason for the exception reported in the `Message` property of the exception object in the `duration` `TextBlock` control. The following code shown in bold highlights the changes you should make:

```
private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
```

```

    try
    {
        await first;
        await second;
        duration.Text = $"Duration (ms): {watch.ElapsedMilliseconds}";
    }
    catch (OperationCanceledException oce)
    {
        duration.Text = oce.Message;
    }
    string message = $"Status of tasks is {first.Status}, {second.Status}";
    ...
}

```

3. In the `generateGraphData` method, comment out the `if` statement that examines the `IsCancellationRequested` property of the `CancellationToken` object and add a statement that calls the `ThrowIfCancellationRequested` method, as shown here in bold:

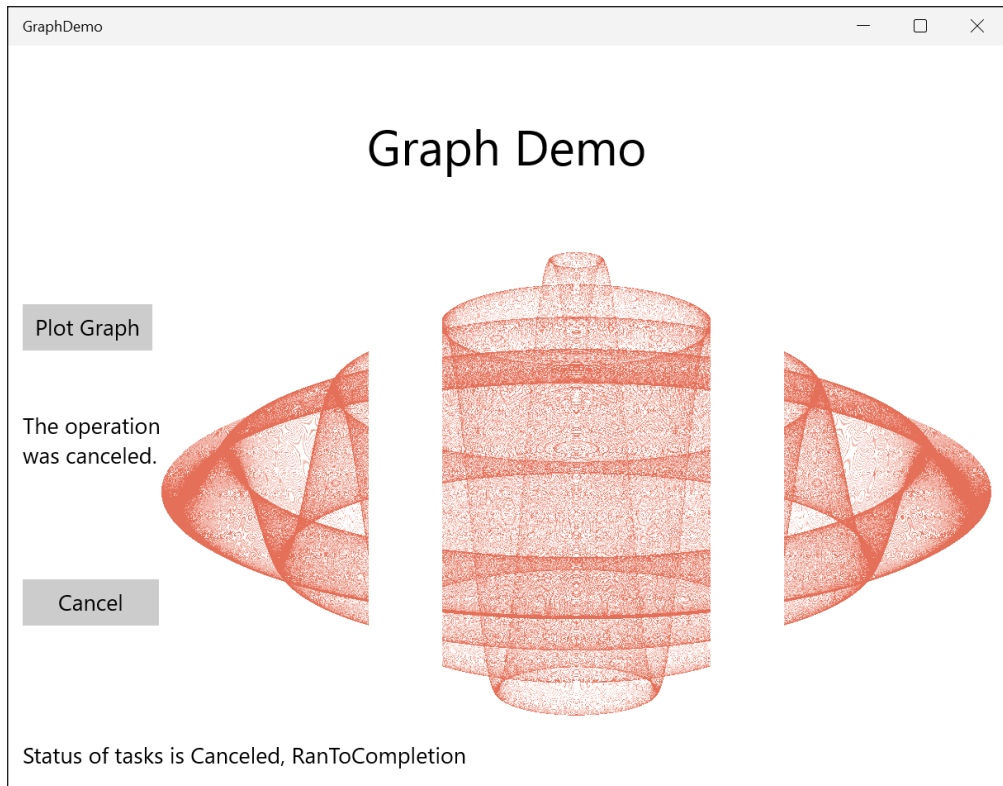
```

private void generateGraphData(byte[] data, int partitionStart, int partitionEnd,
CancellationToken token)
{
    ...
    for (int x = partitionStart; x < partitionEnd; x++);
    {
        ...
        for (double i = -p; i < p; i += 3)
        {
            //if (token.IsCancellationRequested)
            //{
            //    return;
            //}
            token.ThrowIfCancellationRequested();
            ...
        }
    }
    ...
}

```

4. On the **Debug** menu, select **Start Without Debugging**.
5. In the Graph Demo window, select **Plot Graph**, wait for the graph to appear, and verify that the status of both tasks is reported as `RanToCompletion` and the graph is generated.
6. Select **Plot Graph** again, and then quickly select **Cancel**.

If you're quick, the status of one or both tasks should be reported as `Canceled`, the `duration` `TextBox` control should display the text `The operation was canceled`, and the graph should be displayed with holes. If you weren't quick enough, repeat this step to try again. In the following image, one of the tasks actually completed, and the other was canceled. If you are sufficiently speedy, you should be able to see both tasks being canceled.



7. Close the GraphDemo application and return to Visual Studio.

Handling task exceptions by using the `AggregateException` class

You've seen throughout this book that exception handling is an important element in any commercial application. The exception-handling constructs you have encountered so far are straightforward, and if you use them carefully, it's a simple matter to trap an exception and determine which piece of code raised it.

When you start dividing work into multiple concurrent tasks, tracking and handling exceptions becomes a more complex problem. The previous exercise showed how you could catch the `OperationCanceledException` exception that is thrown when you cancel a task. However, there are plenty of other exceptions that might also occur, and different tasks might each generate their own exceptions. Therefore, you need a way to catch and handle multiple exceptions that might be thrown concurrently.

If you're using one of the `Task` wait methods to wait for multiple tasks to complete (using the instance `Wait` method or the static `Task.WaitAll` and `Task.WaitAny` methods), any exceptions thrown by the methods that these tasks are running are gathered into a single exception referred to as an `AggregateException` exception. An `AggregateException` exception acts as a wrapper for a collection of exceptions. Each of the exceptions in the collection might be thrown by different tasks.

In your application, you can catch the `AggregateException` exception and then iterate through this collection and perform any necessary processing.

To help you, the `AggregateException` class provides the `Handle` method. The `Handle` method takes a `Func<Exception, bool>` delegate, which references a method that takes an `Exception` object as its parameter and returns a `Boolean` value. When you call `Handle`, the referenced method runs for each exception in the collection in the `AggregateException` object. The referenced method can examine the exception and take the appropriate action. If the referenced method handles the exception, it should return `true`. If not, it should return `false`. When the `Handle` method completes, any unhandled exceptions are bundled together into a new `AggregateException` exception, and this exception is thrown. A subsequent outer exception handler can then catch this exception and process it.

The code fragment that follows shows an example of a method that can be registered with an `AggregateException` exception handler. This method simply displays the message `Division by zero` occurred if it detects a `DivideByZeroException` exception, or the message `Array index out of bounds` if an `IndexOutOfRangeException` exception occurs. Any other exceptions are left unhandled.

```
private bool handleException(Exception e)
{
    if (e is DivideByZeroException)
    {
        displayErrorMessage("Division by zero occurred");
        return true;
    }
    if (e is IndexOutOfRangeException)
    {
        displayErrorMessage("Array index out of bounds");
        return true;
    }
    return false;
}
```

When you use one of the `Task` wait methods, you can catch the `AggregateException` exception and register the `handleException` method, like this:

```
try
{
    Task first = Task.Run(...);
    Task second = Task.Run(...);
    Task.WaitAll(first, second);
}
catch (AggregateException ae)
{
    ae.Handle(handleException);
}
```

If any of the tasks generate a `DivideByZeroException` or `IndexOutOfRangeException` exception, the `handleException` method will display an appropriate message and acknowledge the exception as handled. Any other exceptions are classified as unhandled and will propagate out from the `AggregateException` exception handler in the customary manner.

There's one additional complication that you should be aware of: when you cancel a task, you've seen that the CLR throws an `OperationCanceledException` exception, and this is the exception that's reported if you're using the `await` operator to wait for the task. However, if you're using one of the Task wait methods, this exception is transformed into a `TaskCanceledException` exception. This is the type of exception that you should be prepared to handle in the `AggregateException` exception handler.

Using continuations with canceled and faulted tasks

If you must perform additional work when a task is canceled or raises an unhandled exception, you can use the `ContinueWith` method with the appropriate `TaskContinuationOptions` value. For example, the following code creates a task that runs the method `doWork`. If the task is canceled, the `ContinueWith` method specifies that another task should be created and run the method `doCancellationWork`. This method can perform some simple logging or tidying up. If the task isn't canceled, the continuation does not run.

```
Task task = new Task(doWork);
task.ContinueWith(doCancellationWork, TaskContinuationOptions.OnlyOnCanceled);
task.Start();
...
private void doWork()
{
    // The task runs this code when it is started
    ...
}
...
private void doCancellationWork(Task task)
{
    // The task runs this code when doWork completes
    ...
}
```

Similarly, you can specify the value `TaskContinuationOptions.OnlyOnFaulted` to specify a continuation that runs if the original method run by the task raises an unhandled exception.

Summary

In this chapter, you learned why it's important to write applications that can scale across multiple processors and processor cores. You saw how to use the `Task` class to run operations in parallel and how to synchronize concurrent operations and wait for them to complete. You learned how to use the `Parallel` class to parallelize some common programming constructs, and you also saw when it is inappropriate to parallelize code. You used tasks and threads together in a graphical user interface to improve responsiveness and throughput, and you saw how to cancel tasks in a clean and controlled manner.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 24.
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

To	Do this
Create a task and run it	Use the static <code>Run</code> method of the <code>Task</code> class to create and run the task in a single step. For example: <pre>Task task = Task.Run(() => doWork()); ... private void doWork() { // The task runs this code when it is started ... }</pre> Or, create a new <code>Task</code> object that references a method to run and call the <code>Start</code> method. For example: <pre>Task task = new Task(doWork); task.Start();</pre>
Wait for a task to finish	Call the <code>Wait</code> method of the <code>Task</code> object. For example: <pre>Task task = ...; ... task.Wait();</pre> Or, use the <code>await</code> operator (only in an <code>async</code> method). For example: <pre>await task;</pre>
Wait for several tasks to finish	Call the static <code>WaitAll</code> method of the <code>Task</code> class, and specify the tasks to wait for. For example: <pre>Task task1 = ...; Task task2 = ...; Task task3 = ...; Task task4 = ...; ... Task.WaitAll(task1, task2, task3, task4);</pre>
Specify a method to run in a new task when a task has completed	Call the <code>ContinueWith</code> method of the task and specify the method as a continuation. For example: <pre>Task task = new Task(doWork); task.ContinueWith(doMoreWork, TaskContinuationOptions.NotOnFaulted);</pre>
Perform loop iterations and statement sequences by using parallel tasks	Use the <code>Parallel.For</code> and <code>Parallel.ForEach</code> methods to perform loop iterations by using tasks. For example: <pre>Parallel.For(0, 100, performLoopProcessing); ... private void performLoopProcessing(int x) { // Perform loop processing }</pre> Or, use the <code>Parallel.Invoke</code> method to perform concurrent method calls by using separate tasks. For example: <pre>Parallel.Invoke(doWork, doMoreWork, doYetMoreWork);</pre>

To	Do this
Handle exceptions raised by one or more tasks	Catch the <code>AggregateException</code> exception. Use the <code>Handle</code> method to specify a method that can handle each exception in the <code>AggregateException</code> object. If the exception-handling method handles the exception, return <code>true</code>; otherwise, return <code>false</code>. For example: <pre> try { Task task = Task.Run(...); task.Wait(); ... } catch (AggregateException ae) { ae.Handle(handleException); } ... private bool handleException(Exception e) { if (e is TaskCanceledException) { ... return true; } else { return false; } } </pre>
Enable cancellation in a task	Implement cooperative cancellation by creating a <code>CancellationTokenSource</code> object and using a <code>CancellationToken</code> parameter in the method run by the task. In the task method, call the <code>ThrowIfCancellationRequested</code> method of the <code>CancellationToken</code> parameter to throw an <code>OperationCanceledException</code> exception and terminate the task. For example: <pre> private void generateGraphData(..., CancellationToken token) { ... token.ThrowIfCancellationRequested(); ... } </pre>

Improving response time by performing asynchronous operations

After completing this chapter, you will be able to:

- Define and use asynchronous methods to improve the interactive response time of applications that perform long-running and I/O-bound operations.
- Explain how to reduce the time taken to perform complex LINQ queries by using parallelization.
- Use the concurrent collection classes to safely share data between parallel tasks.

Chapter 23, “Improving throughput by using tasks,” demonstrated how to use the `Task` class to perform operations in parallel and improve throughput in compute-bound applications. However, although maximizing the processing power available to an application can make it run more quickly, responsiveness is also important. Recall that the Windows user interface operates by using a single thread of execution, but users expect an application to respond when they select a button on a form—even if the application is currently performing a large and complex calculation. Additionally, some tasks might take a considerable time to run even if they’re not compute-bound—for example, an I/O-bound task waiting to receive information across the network from a remote website—and blocking user interaction while waiting for an event that might take an indeterminate time to happen is clearly not good design practice. The solution to both of these problems is the same: perform the task asynchronously and leave the user interface thread free to handle user interactions.

Issues with response time are not limited to user interfaces. For example, Chapter 21, “Querying in-memory data by using query expressions,” shows how you can access data held in memory in a declarative manner by using Language-Integrated Query (LINQ). A typical LINQ query generates an enumerable result set, and you can iterate serially through this set to retrieve the data. If the data source used to generate the result set is large, running a LINQ query can take a long time. Many database-management systems faced with the issue of optimizing queries address this issue by using algorithms that break down the process of identifying the data for a query into a series of tasks. They then run these tasks in parallel, combining the results when the tasks have completed to generate the complete result set. The designers of .NET decided to provide LINQ with a similar facility, and the result is Parallel LINQ (PLINQ). You will study PLINQ in the second part of this chapter.

Asynchronicity and scalability

Asynchronicity is a powerful concept that you need to understand if you're building large-scale solutions such as enterprise web applications and services. A web server typically has limited resources for handling requests from a potentially very large audience, each member of which expects his or her requests to be handled quickly. In many cases, a user request can invoke a series of operations that individually can take significant time (perhaps as much as a second or two). Consider an e-commerce system in which a user is querying the product catalog or placing an order, for example. Both of these operations typically involve reading and writing data held in a database that might be managed by a database server remote from the web server. Many web servers can support only a limited number of concurrent connections, and if the thread associated with a connection is waiting for an I/O operation to complete, that connection is effectively blocked. If the thread creates a separate task to handle the I/O asynchronously, then the thread can be released and the connection recycled for another user. This approach is far more scalable than implementing such operations synchronously.

For an example and a detailed explanation of why performing synchronous I/O is bad in this situation, read about the Synchronous I/O anti-pattern in the public Microsoft Patterns & Practices Git repository at <https://github.com/mspnp/performance-optimization/tree/master/SynchronousIO>.

Implementing asynchronous methods

An *asynchronous* method is one that does not block the current thread on which it starts to run. When an application invokes an asynchronous method, an implied contract expects the method to return control to the calling environment quite quickly and to perform its work on a separate thread. The definition of *quite* is not a mathematically defined quantity, but the expectation is that if an asynchronous method performs an operation that might cause a noticeable delay to the caller, it should do so by using a background thread, enabling the caller to continue running on the current thread. This process sounds complicated, and indeed in earlier versions of .NET, it was. However, C# now provides the `async` method modifier and the `await` operator, which abstract much of this complexity to the compiler—meaning that (most of the time) you no longer have to concern yourself with the intricacies of multithreading.

Defining asynchronous methods: the problem

You've already seen how you can implement concurrent operations by using `Task` objects. To quickly recap, when you initiate a task by using the `Start` or `Run` method of the `Task` type, the common language runtime (CLR) uses its own scheduling algorithm to allocate the task to a thread and run this thread at a time convenient to the operating system when sufficient resources are available. This approach frees your code from the requirement to recognize and manage the workload of your

computer. If you need to perform another operation when a specific task completes, you have the following choices:

- You can manually wait for the task to finish by using one of the `Wait` methods exposed by the `Task` type. You can then initiate the new operation, possibly by defining another task.
- You can define a continuation. A *continuation* simply specifies an operation to be performed when a given task completes. The .NET runtime automatically executes the continuation operation as a task that it schedules when the original task finishes. The continuation reuses the same thread as the original task.

However, even though the `Task` type provides a convenient generalization of an operation, you still often have to write potentially awkward code to solve some of the common problems that developers encounter when using a background thread. For example, suppose that you define the following method, which performs a series of long-running operations that must run in a serial manner and then displays a message in a `TextBox` control on the screen:

```
private void slowMethod()
{
    doFirstLongRunningOperation();
    doSecondLongRunningOperation();
    doThirdLongRunningOperation();
    message.Text = "Processing Completed";
}

private void doFirstLongRunningOperation()
{
    ...
}

private void doSecondLongRunningOperation()
{
    ...
}

private void doThirdLongRunningOperation()
{
    ...
}
```

If you invoke `slowMethod` from a piece of user interface code (such as the `Click` event handler for a button control), the user interface will become unresponsive until this method completes. You can make the `slowMethod` method more responsive by using a `Task` object to run the `doFirstLongRunningOperation` method and define continuations for the same `Task` that run the `doSecondLongRunningOperation` and `doThirdLongRunningOperation` methods in turn, like this:

```
private void slowMethod()
{
    Task task = new Task(doFirstLongRunningOperation);
    task.ContinueWith(doSecondLongRunningOperation);
    task.ContinueWith(doThirdLongRunningOperation);
    task.Start();
    message.Text = "Processing Completed"; // When does this message appear?
}
```

```
private void doFirstLongRunningOperation()
{
    ...
}

private void doSecondLongRunningOperation(Task t)
{
    ...
}

private void doThirdLongRunningOperation(Task t)
{
    ...
}
```

Although this refactoring seems fairly simple, you should note a few points. Specifically, the signatures of the `doSecondLongRunningOperation` and `doThirdLongRunningOperation` methods have changed to accommodate the requirements of continuations. (The `Task` object that instigated the continuation is passed as a parameter to a continuation method.) More importantly, you need to ask yourself, “When is the message displayed in the `TextBox` control?” The issue with this second point is that although the `Start` method initiates a `Task` operation, it does not wait for it to complete, so the message appears while the processing is being performed rather than when it has finished.

This is a somewhat trivial example, but the general principle is important, and there are at least two solutions. The first is to wait for the `Task` operation to complete before displaying the message, like this:

```
private void slowMethod()
{
    Task task = new Task(doFirstLongRunningOperation);
    task.ContinueWith(doSecondLongRunningOperation);
    task.ContinueWith(doThirdLongRunningOperation);
    task.Start();
    task.Wait();
    message.Text = "Processing Completed";
}
```

However, the call to the `Wait` method now blocks the thread executing the `slowMethod` method and obviates the purpose of using a `Task` operation in the first place.



Important Generally speaking, you should *never* call the `Wait` method directly in the user interface thread.

A better solution is to define a continuation that displays the message and arrange for it to run only when the `doThirdLongRunningOperation` method finishes, in which case you can remove the call to the `Wait` method. You might be tempted to implement this continuation as a delegate as shown in bold in the following code (remember that a continuation is passed a `Task` object as an argument; that is the purpose of the `t` parameter to the delegate):

```
private void slowMethod()
{
    Task task = new Task(doFirstLongRunningOperation);
    task.ContinueWith(doSecondLongRunningOperation);
    task.ContinueWith(t => { message.Text = "Processing Completed"; }, task);
}
```

```

        task.ContinueWith(doThirdLongRunningOperation);
        task.ContinueWith((t) => message.Text = "Processing Complete");
        task.Start();
    }

```

Unfortunately, this approach exposes another problem. If you try to run this code in debug mode, you'll find that the final continuation generates a `System.Exception` exception with the rather obscure message: The application called an interface that was marshaled for a different thread. The issue here is that only the user interface thread can manipulate user interface controls, and now you're attempting to write to a `TextBox` control from a different thread—the thread being used to run the `Task` operation.

You can resolve this problem by using the `Dispatcher` object. The `Dispatcher` object is a component of the user interface infrastructure, and you can send it requests to perform work on the user interface thread by calling its `RunAsync` method. This method takes an `Action` delegate that specifies the code to run. The details of the `Dispatcher` object and the `RunAsync` method are beyond the scope of this book, but the following example shows how you might use them to display the message required by the `slowMethod` method from a continuation:

```

private void slowMethod()
{
    Task task = new Task(doFirstLongRunningOperation);
    task.ContinueWith(doSecondLongRunningOperation);
    task.ContinueWith(doThirdLongRunningOperation);
    task.ContinueWith((t) => this.Dispatcher.RunAsync(
        CoreDispatcherPriority.Normal,
        () => message.Text = "Processing Complete"));
    task.Start();
}

```

This works, but it's messy and difficult to maintain. You now have a delegate (the continuation) specifying another delegate (the code to be run by `RunAsync`).



More Info You can find more information about the `Dispatcher` object and the `RunAsync` method on the Microsoft website at <https://msdn.microsoft.com/library/windows.ui.core.coredispatcher.runasync>.

Defining asynchronous methods: the solution

The purpose of the `async` and `await` keywords in C# is to enable you to define and call methods that can run asynchronously. This means that you don't have to concern yourself with specifying continuations or scheduling code to run on `Dispatcher` objects to ensure that data is manipulated on the correct thread. Very simply:

- The `async` modifier indicates that a method contains functionality that can be run asynchronously.
- The `await` operator specifies the points at which this asynchronous functionality should be performed.

The following code example shows the `slowMethod` method implemented as an asynchronous method with the `async` modifier and `await` operators:

```
private async void slowMethod()
{
    await doFirstLongRunningOperation();
    await doSecondLongRunningOperation();
    await doThirdLongRunningOperation();
    message.Text = "Processing Complete";
}
```

This method now looks remarkably similar to the original version, and that is the power of `async` and `await`. In fact, this magic is nothing more than an exercise in reworking your code by the C# compiler. When the C# compiler encounters the `await` operator in an `async` method, it effectively reformats the operand that follows this operator as a task that runs on the same thread as the `async` method. The remainder of the code is converted into a continuation that runs when the task completes, again running on the same thread. Now, because the thread that was running the `async` method was the thread running the user interface, it has direct access to the controls in the window, which means it can update them directly without routing them through the `Dispatcher` object.

Although this approach looks quite simple at first glance, keep the following points in mind to avoid some possible misconceptions:

- The `async` modifier does not signify that a method runs asynchronously on a separate thread. All it does is specify that the code in the method can be divided into one or more continuations. When these continuations run, they execute on the same thread as the original method call.
- The `await` operator specifies a point at which the C# compiler can split the code into a continuation. The `await` operator itself expects its operand to be an awaitable object. An *awaitable object* is a type that provides the `GetAwaiter` method, which returns an object that in turn provides methods for running code and waiting for it to complete. The C# compiler converts your code into statements that use these methods to create an appropriate continuation.



Important You can use the `await` operator only in a method marked with `async`. Outside an `async` method, the `await` keyword is treated as an ordinary identifier. (You can even create a variable called `await`, although this is not recommended.)



Note Prior to C# 7.1, the `Main` method could not be marked as `async`. This restriction has since disappeared. If you're running .NET 6.0 or later, you can tag `Main` as `async`, and use the `await` operator to run asynchronous methods. The `Main` method only terminates when the asynchronous methods complete.

In the current implementation of the `await` operator, the awaitable object it expects you to specify as the operand is a `Task` object. This means that you must make some modifications to the `doFirstLongRunningOperation`, `doSecondLongRunningOperation`, and `doThirdLongRunningOperation` methods. Specifically, each method must now create and run a `Task` object to perform

its work and return a reference to this Task. The following example shows an amended version of the `doFirstLongRunningOperation` method:

```
private Task doFirstLongRunningOperation()
{
    Task t = Task.Run(() => { /* original code for this method goes here */ });
    return t;
}
```

It's also worth considering whether there are opportunities to break the work done by the `doFirstLongRunningOperation` method into a series of parallel operations. If so, you can divide the work into a set of Task operations, as described in Chapter 23. However, which of these Task objects should you return as the result of the method?

```
private Task doFirstLongRunningOperation()
{
    Task first = Task.Run(() => { /* code for first operation */ });
    Task second = Task.Run(() => { /* code for second operation */ });
    return ...; // Do you return first or second?
}
```

If the method returns `first`, the `await` operator in the `slowMethod` will wait only for that Task operation to complete and not for the second one. Similar logic applies if the method returns `second`. The solution is to define the `doFirstLongRunningOperation` method with `async` and `await` each of the Task operations, as shown here:

```
private async Task doFirstLongRunningOperation()
{
    Task first = Task.Run(() => { /* code for first operation */ });
    Task second = Task.Run(() => { /* code for second operation */ });
    await first;
    await second;
}
```

Remember that when the compiler encounters the `await` operator, it generates code that waits for the item specified by the argument to complete with a continuation that runs the statements that follow. You can think of the value returned by the `async` method as a reference to the Task operation that runs this continuation. (This description is not completely accurate, but it's a good enough model for the purposes of this chapter.) So, the `doFirstLongRunningOperation` method creates and starts the tasks `first` and `second` running in parallel, the compiler reformats the `await` statements into code that waits for `first` to complete followed by a continuation that waits for `second` to finish, and the `async` modifier causes the compiler to return a reference to this continuation. Notice that because the compiler now determines the return value of the method, you no longer specify a return value yourself. (In fact, if you try to return a value, in this case, your code will not compile.)



Note If you don't include an `await` statement in an `async` method, the method is simply a reference to a Task operation that performs the code in the body of the method. As a result, when you invoke the method, it doesn't actually run asynchronously. In this case, the compiler will warn you with the message, "This `async` method lacks `await` operators and will run synchronously."



Tip You can use the `async` modifier to prefix a delegate, making it possible to create delegates that incorporate asynchronous processing by using the `await` operator.

In the following exercise, you'll modify the `GraphDemo` application from Chapter 23 to generate the data for the graph by using an asynchronous method.

To modify the `GraphDemo` application to use an asynchronous method

1. Using Microsoft Visual Studio 2022, open the **GraphDemo** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 24\GraphDemo** folder in your **Documents** folder.
2. In Solution Explorer, expand the **MainPage.xaml** node and open the **MainPage.xaml.cs** file in the Code and Text Editor window.
3. In the `MainPage` class, locate the `plotButton_Click` method. The code in this method looks like this:

```
private void plotButton_Click(object sender, RoutedEventArgs e)
{
    try
    {
        Random rand = new Random();
        redValue = (byte)rand.Next(0xFF);
        greenValue = (byte)rand.Next(0xFF);
        blueValue = (byte)rand.Next(0xFF);
        tokenSource = new CancellationTokenSource();
        CancellationToken token = tokenSource.Token;

        int dataSize = bytesPerPixel * pixelWidth * pixelHeight;
        byte[] data = new byte[dataSize];

        Stopwatch watch = Stopwatch.StartNew();

        try
        {
            generateGraphData(data, 0, pixelWidth / 2, token);
            duration.Text = $"Duration (ms): {watch.ElapsedMilliseconds}";
        }
        catch (OperationCanceledException oce)
        {
            duration.Text = oce.Message;
        }
        duration.Text = $"Duration (ms): {watch.ElapsedMilliseconds}";

        WriteableBitmap graphBitmap = new WriteableBitmap(pixelWidth, pixelHeight);

        using (Stream pixelStream = graphBitmap.PixelBuffer.AsStream())
        {
            pixelStream.Seek(0, SeekOrigin.Begin);
            pixelStream.Write(data, 0, data.Length);
            graphBitmap.Invalidate();
            graphImage.Source = graphBitmap;
        }
    }
}
```

```

    }
    catch (Exception ex)
    {
        MessageDialog msg = new MessageDialog("Exception", ex.Message);
        msg.ShowAsync();
    }
}

```

This is a simplified version of the application from the previous chapter. It invokes the `generateGraphData` method directly from the user interface thread and does not use Task objects to generate the data for the graph in parallel.



Note If you reduced the size of the `pixelWidth` and `pixelHeight` fields in the exercises in Chapter 23 to save memory, do so again in this version before proceeding with the next step.

4. In the Visual Studio toolbar, make sure the platform is set to **x64**. Then, on the **Debug** menu, select **Start Without Debugging**.
5. In the GraphDemo window, select **Plot Graph**. While the data is being generated, select **Cancel**.

Notice that the user interface is completely unresponsive as the graph is being generated and displayed. This is because the `plotButton_Click` method performs all its work synchronously, including the generation of the data for the graph.

6. Close the GraphDemo application and return to Visual Studio.
7. In the Code and Text Editor window displaying the `MainPage` class, above the `generateGraphData` method, add a new private method called `generateGraphDataAsync`. This method should take the same list of parameters as the `generateGraphData` method, but it should return a Task object rather than a void. The method should also be marked with `async` and should look like this:

```

private async Task generateGraphDataAsync(byte[] data,
    int partitionStart, int partitionEnd,
    CancellationToken token)
{
}

```



Note It's recommended practice to name asynchronous methods with the `Async` suffix.

8. In the `generateGraphDataAsync` method, add the statements shown here in bold.

```

private async Task generateGraphDataAsync(byte[] data, int
partitionStart, int partitionEnd, CancellationToken token)
{
    Task task = Task.Run(() =>
        generateGraphData(data, partitionStart, partitionEnd, token));
    await task;
}

```

This code creates a Task object that runs the generateGraphData method and uses the await operator to wait for the Task operation to complete. The Task object generated by the compiler as a result of the await operator is the value returned from the method.

9. Return to the plotButton_Click method and change the definition of this method to include the async modifier, as shown in bold in the following code:

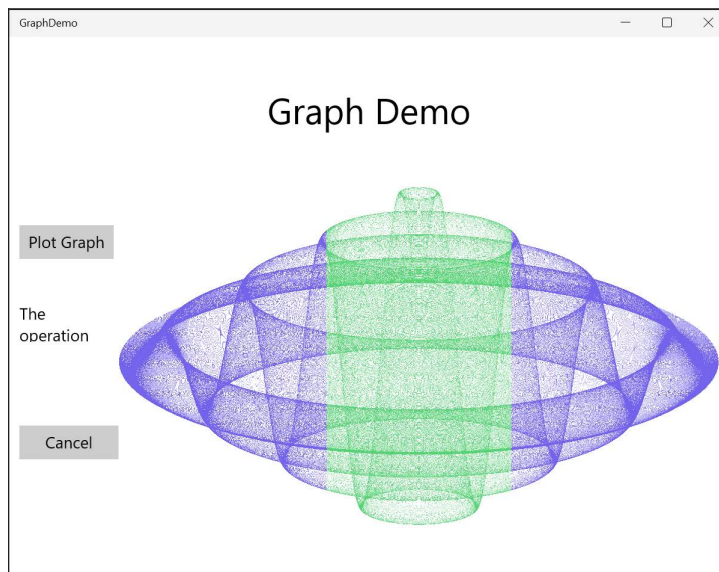
```
private async void plotButton_Click(object sender, RoutedEventArgs e)
{
    ...
}
```

10. In the inner try block in the plotButton_Click method, modify the statement that generates the data for the graph to call the generateGraphDataAsync method asynchronously, as shown here in bold:

```
try
{
    await generateGraphDataAsync(data, 0, pixelWidth / 2, token);
    duration.Text = $"Duration (ms): {watch.ElapsedMilliseconds}";
}
...
```

11. On the **Debug** menu, select **Start Without Debugging**.
12. In the GraphDemo window, select **Plot Graph** and verify that the application generates the graph correctly.
13. Select **Plot Graph** again. Then, while the data is being generated, select **Cancel**.

This time, the user interface should be responsive. Only part of the graph should be generated.



14. Close the GraphDemo application and return to Visual Studio.

Defining asynchronous methods that return values

So far, all the examples you've seen use a `Task` object to perform a piece of work that doesn't return a value. However, you also use tasks to run methods that calculate a result. To do this, you use the generic `Task<TResult>` class, where the type parameter, `TResult`, specifies the type of the result.

You create and start a `Task<TResult>` object in much the same way you do an ordinary `Task` object. The primary difference is that the code you execute should return a value. For example, the method named `calculateValue` shown in the code example that follows generates an integer result. To invoke this method by using a task, you create and run a `Task<int>` object. You obtain the value returned by the method by querying the `Result` property of the `Task<int>` object. If the task hasn't finished running the method and the result is not yet available, the `Result` property blocks the caller. This means that you don't have to perform any synchronization yourself, and you know that when the `Result` property returns a value, the task has completed its work.

```
Task<int> calculateValueTask = Task.Run(() => calculateValue(...));
...
int calculatedData = calculateValueTask.Result; // Block until calculateValueTask completes
...
private int calculateValue(...)
{
    int someValue;
    // Perform calculation and populate someValue
    ...
    return someValue;
}
```

The generic `Task<TResult>` type is also the basis of the mechanism for defining asynchronous methods that return values. In previous examples, you saw that you implement asynchronous void methods by returning a `Task`. If an asynchronous method actually generates a result, it should return a `Task<TResult>` object. This is shown in the following example, which creates an asynchronous version of the `calculateValue` method:

```
private async Task<int> calculateValueAsync(...)
{
    // Invoke calculateValue using a Task
    Task<int> generateResultTask = Task.Run(() => calculateValue(...));
    await generateResultTask;
    return generateResultTask.Result;
}
```

This method looks slightly confusing because the return type is specified as `Task<int>`, but the return statement actually returns an `int`. Remember that when you define an `async` method, the compiler performs some refactoring of your code, and it essentially returns a reference to `Task` that runs the continuation for the statement `return generateResultTask.Result`; . The type of the expression returned by this continuation is `int`, so the return type of the method is `Task<int>`.

To invoke an asynchronous method that returns a value, use the `await` operator, like this:

```
int result = await calculateValueAsync(...);
```

The `await` operator extracts the value from the `Task` object returned by the `calculateValueAsync` method, and in this case assigns it to the `result` variable.

Asynchronous method pitfalls

The `async` and `await` operators have been known to cause confusion among programmers. It's important to understand that:

- Marking a method as `async` does not mean that it runs asynchronously. It means that the method can contain statements that may run asynchronously.
- The `await` operator indicates that a method should be run by a separate task and the calling code is suspended until the method call completes. The thread used by the calling code is released and can be reused. This is important if the thread is the user interface thread, as it enables the user interface to remain responsive.
- Using the `await` operator isn't the same as using the `Wait` method of a task. The `Wait` method always blocks the current thread and doesn't allow it to be reused until the task completes.
- By default, after an `await` operator, the code that resumes execution attempts to obtain the original thread that was used to invoke the asynchronous method call. If this thread is busy, the code will be blocked. You can use the `ConfigureAwait(false)` method to specify that the operation can be resumed on any available thread and reduce the chances of blocking. This is especially useful for web applications and services that may need to handle many thousands of concurrent requests.
- You shouldn't use `ConfigureAwait(false)` if the code that runs after an `await` operator must execute on the original thread. In the example discussed earlier, adding `ConfigureAwait(false)` to each awaited operation will result in the likelihood that the continuations the compiler generates will run on separate threads. This includes the continuation that attempts to set the `Text` property for `message`, causing the display of the following message: The application called an interface that was marshaled for a different thread.

```
private async void slowMethod()
{
    await doFirstLongRunningOperation().ConfigureAwait(false);
    await doSecondLongRunningOperation().ConfigureAwait(false);
    await doThirdLongRunningOperation().ConfigureAwait(false);
    message.Text = "Processing Complete";
}
```

- Careless use of asynchronous methods that return results and that run on the user interface thread can generate deadlocks, causing the application to freeze. Consider the following example:

```
private async void myMethod()
{
    var data = generateResult();
    ...
    message.Text = $"result: {data.Result}";
}

private async Task<string> generateResult()
{
    string result;
    ...
}
```

```

        result = ...
        return result;
    }

```

In this code, the `generateResult` method returns a string value. However, the `myMethod` method doesn't actually start the task that runs the `generateResult` method until it attempts to access the `data.Result` property. The `data` object is a reference to the task, and if the `Result` property is not available because the task has not been run, then accessing this property will block the current thread until the `generateResult` method completes. Furthermore, the task used to run the `generateResult` method attempts to resume the thread on which it was invoked when the method completes (the user interface thread), but this thread is now blocked. The result is that the `myMethod` method can't finish until the `generateResult` method completes, and the `generateResult` method can't finish until the `myMethod` method completes. The solution to this problem is to await the task that runs the `generateResult` method. You can do this as follows:

```

private async void myMethod()
{
    var data = generateResult();
    ...
    message.Text = $"result: {await data}";
}

```

Asynchronous methods and the Windows Runtime APIs

The designers of Windows 8 and later versions wanted to ensure that applications were as responsive as possible, so when they implemented the Windows Runtime (WinRT), they decided that any operation that might take more than 50 milliseconds to perform should be available only through an asynchronous API. You might have noticed one or two instances of this approach already in this book. For example, to display a message to a user, you can use a `MessageDialog` object. However, when you display this message, you must use the `ShowAsync` method, like this:

```

using Windows.UI.Popups;
...
MessageDialog dlg = new MessageDialog("Message to user");
await dlg.ShowAsync();

```

The `MessageDialog` object displays the message and waits for the user to select the Close button that appears as part of this dialog. Any form of user interaction might take an indeterminate length of time—the user might have gone for lunch before selecting Close—and it's often important not to block the application or prevent it from performing other operations (such as responding to events) while the dialog is displayed. The `MessageDialog` class doesn't provide a synchronous version of the `ShowAsync` method, but if you need to display a dialog synchronously, you can simply call `dlg.ShowAsync()` without the `await` operator.

Another common example of asynchronous processing concerns the `FileOpenPicker` class, which you saw in Chapter 5, "Using compound assignment and iteration statements." The `FileOpenPicker` class displays a list of files that the user can select from. As with the `MessageDialog` class, the user might take a considerable time browsing and selecting files, so this operation should not block the

application. The following example shows how to use the `FileOpenPicker` class to display the files in the user's Documents folder and wait while the user selects a single file from this list:

```
using Windows.Storage;
using Windows.Storage.Pickers;
...
FileOpenPicker fp = new FileOpenPicker();
fp.SuggestedStartLocation = PickerLocationId.DocumentsLibrary;
fp.ViewMode = PickerViewMode.List;
fp.FileTypeFilter.Add("*");
StorageFile file = await fp.PickSingleFileAsync();
```

The key statement is the line that calls the `PickSingleFileAsync` method. This is the method that displays the list of files and allows the user to navigate around the file system and select a file. (The `FileOpenPicker` class also provides the `PickMultipleFilesAsync` method by which a user can select more than one file.) The value returned by this method is `Task<StorageFile>`, and the `await` operator extracts the `StorageFile` object from this result. The `StorageFile` class provides an abstraction of a file held on hard disk, and by using this class, you can open a file and read from it or write to it.



Note Strictly speaking, the `PickSingleFileAsync` method returns an object of type `IAsyncOperation<StorageFile>`. WinRT uses its own abstraction of asynchronous operations and maps .NET Task objects to this abstraction; the Task class implements the `IAsyncOperation` interface. If you're programming in C#, your code is not affected by this transformation, and you can simply use Task objects without concerning yourself with how they get mapped to WinRT asynchronous operations.

File input/output (I/O) is another source of potentially slow operations. The `StorageFile` class implements several asynchronous methods by which these operations can be performed without affecting the responsiveness of an application. For example, in Chapter 5, after the user selects a file using a `FileOpenPicker` object, the code then opens this file for reading, asynchronously:

```
StorageFile file = await fp.PickSingleFileAsync();
...
var fileStream = await file.OpenAsync(FileAccessMode.Read);
```

One final example that's directly applicable to the exercises you've seen in this chapter and the previous one concerns writing to a stream. You might have noticed that although the time reported to generate the data for the graph is a few seconds, it can take up to twice that long for the graph to actually appear. This happens because of the way the data is written to the bitmap. The bitmap renders data held in a buffer as part of the `WriteableBitmap` object, and the `AsStream` extension method provides a `Stream` interface to this buffer. The data is written to the buffer via this stream by using the `Write` method, like this:

```
...
Stream pixelStream = graphBitmap.PixelBuffer.AsStream();
pixelStream.Seek(0, SeekOrigin.Begin);
pixelStream.Write(data, 0, data.Length);
...
```

Unless you've reduced the value of the `pixelWidth` and `pixelHeight` fields to save memory, the volume of data written to the buffer is just over 570 MB (15,000 * 10,000 * 4 bytes), so this `Write` operation can take a few seconds. To improve response time, you can perform this operation asynchronously by using the `WriteAsync` method:

```
await pixelStream.WriteAsync(data, 0, data.Length);
```

In summary, when you build applications for Windows, you should seek to exploit asynchronicity wherever possible.

Tasks, memory allocation, and efficiency

Just because a method is tagged as `async`, it doesn't mean that it will always execute asynchronously. Consider the following method:

```
public async Task<int> FindValueAsync(string key)
{
    bool foundLocally = GetCachedValue(key, out int result);
    if (foundLocally)
        return result;

    result = await RetrieveValue(key); // possibly takes a long time
    AddItemToLocalCache(key, result);
    return result;
}
```

The purpose of this method is to look up an integer value associated with a string key. For example, you might be looking up a customer ID based on the customer's name, or you could be retrieving a piece of data based on a string containing an encrypted key. The `FindValueAsync` method implements the Cache-Aside pattern (see <https://docs.microsoft.com/en-us/azure/architecture/patterns/cache-aside> for a detailed discussion of this pattern), whereby the results of a potentially lengthy calculation or lookup operation are cached locally when they are performed in case they're needed again in the near future. If the same key value is passed to a subsequent call of `FindValueAsync`, the cached data can be retrieved.

The Cache-Aside pattern uses the following helper methods (the implementations of these methods is not shown):

- **GetCachedValue** This method checks the cache for an item with the specified key and passes the item back if it's available in the `out` parameter. The return value of the method is `true` if the data was found in the cache, and `false` otherwise.
- **RetrieveValue** This method runs if the item was not found in the cache. It performs the calculation or lookups necessary to find the data and returns it. This method could potentially take a significant time to run, so it is performed asynchronously.
- **AddItemToLocalCache** This method adds the specified item to the local cache in case it is requested again. This will save the application from having to perform the expensive `RetrieveValue` operation again.

In an ideal world, the cache will account for the vast majority of the data requested over the lifetime of the application, and the number of times it is necessary to invoke `RetrieveValue` should become shrinkingly small.

Now consider what happens each time your code calls the `FindValueAsync` method. In most cases, the work will be performed synchronously. (It finds the data in cache.) The data is an integer, but it's returned wrapped in a `Task<int>` object. Creating and populating this object, and then retrieving the data from this object when the method returns, requires more effort in terms of processing power and memory allocation than simply returning an `int`. C# caters to this situation by providing the `ValueTask` generic type. You use it to specify the return type of an async method, but the return value is marshaled as a value type on the stack rather than a reference on the heap:

```
public async ValueTask<int> FindValueAsync(string key)
{
    bool foundLocally = GetCachedValue(key, out int result);
    if (foundLocally)
        return result;

    result = await RetrieveValue(key); // possibly takes a long time
    AddItemToLocalCache(key, result);
    return result;
}
```

This doesn't mean that you should always use `ValueTask` rather than `Task`. If an asynchronous method actually performs the `await` operation, then using `ValueTask` can decrease the efficiency of your code quite significantly for reasons that I don't have time or space to go into here. So, in general, consider returning a `ValueTask` object only if the vast majority of the calls to an async method are likely to be performed synchronously; otherwise, stick to the `Task` type.



Note To use the `ValueType` type, you must use the NuGet Package Manager to add the `System.Threading.Tasks.Extensions` package to your project.

The `IAsyncResult` design pattern in earlier versions of .NET

Asynchronicity has long been recognized as a key element in building responsive applications with .NET, and the concept predates the introduction of the `Task` class in version 4.0 of the .NET Framework. Microsoft introduced the `IAsyncResult` design pattern based on the `AsyncCallback` delegate type to handle these situations.

The exact details of how this pattern works are beyond the scope of this book, but from a programmer's perspective, the implementation of this pattern meant that many types in the .NET class library exposed long-running operations in two ways:

- In a synchronous form consisting of a single method
- In an asynchronous form that used a pair of methods, named `BeginOperationName` and `EndOperationName`, where `OperationName` specified the operation being performed

For example, the `MemoryStream` class in the `System.IO` namespace provides the `Write` method to write data synchronously to a stream in memory, but it also provides the `BeginWrite` and `EndWrite` methods to perform the same operation asynchronously. The `BeginWrite` method initiates the write operation that's performed on a new thread. The `BeginWrite` method expects the programmer to provide a reference to a callback method that runs when the write operation completes; this reference is in the form of an `AsyncCallback` delegate. In this method, the programmer should implement any appropriate tidying up and call the `EndWrite` method to signify that the operation has completed. The following code example shows this pattern in action:

```
...
Byte[] buffer = ...; // populated with data to write to the MemoryStream
MemoryStream ms = new MemoryStream();
AsyncCallback callback = new AsyncCallback(handleWriteCompleted);
ms.BeginWrite(buffer, 0, buffer.Length, callback, ms);
...
private void handleWriteCompleted(IAsyncResult ar)
{
    MemoryStream ms = ar.AsyncState as MemoryStream;
    ...// Perform any appropriate tidying up
    ms.EndWrite(ar);
}
```

The parameter to the callback method (`handleWriteCompleted`) is an `IAsyncResult` object that contains information about the status of the asynchronous operation and any other state information. You can pass user-defined information to the callback in this parameter; the final argument supplied to the `BeginOperationName` method is packaged into this parameter. In this example, the callback is passed a reference to `MemoryStream`.

Although this sequence works, it's a messy paradigm that obscures the operation you're performing. The code for the operation is split into two methods, and it's easy to lose the mental connection between these methods if you have to maintain this code. If you're using `Task` objects, you can simplify this model by calling the static `FromAsync` method of the `TaskFactory` class. This method takes the `BeginOperationName` and `EndOperationName` methods and wraps them into code that is performed by using a `Task` operation. There's no need to create an `AsyncCallback` delegate because this is generated behind the scenes by the `FromAsync` method. So, you can perform the same operation shown in the previous example like this:

```
...
Byte[] buffer = ...;
MemoryStream s = new MemoryStream();
Task t = Task<int>.Factory.FromAsync(
    s.Beginwrite, s.EndWrite, buffer, 0, buffer.Length, null);
t.Start();
await t;
...
```

This technique is useful if you need to access asynchronous functionality exposed by types developed in earlier versions of .NET.

Using PLINQ to parallelize declarative data access

Data access is another area where response time is important, especially if you're building applications that must search through lengthy data structures. In earlier chapters, you saw how powerful LINQ is for retrieving data from an enumerable data structure, but the examples shown were inherently single-threaded. Parallel LINQ (PLINQ) provides a set of extensions to LINQ that is based on Task objects and that can help you boost performance and parallelize some query operations.

PLINQ works by dividing a data set into partitions and then using tasks to retrieve the data that matches the criteria specified by the query for each partition in parallel. When the tasks have completed, the results retrieved for each partition are combined into a single enumerable result set. PLINQ is ideal for scenarios that involve data sets with large numbers of elements or if the criteria specified for matching data involve operations that are complex and computationally expensive.

An important aim of PLINQ is to be as nonintrusive as possible. To convert a LINQ query into a PLINQ query, you use the `AsParallel` extension method. The `AsParallel` method returns a `ParallelQuery` object that acts similarly to the original enumerable object, except that it provides parallel implementations of many of the LINQ operators, such as `join` and `where`. These implementations of the LINQ operators are based on tasks and use various algorithms to try to run parts of your LINQ query in parallel wherever possible. However, as ever in the world of parallel computing, the `AsParallel` method is not magic. You can't guarantee that your code will speed up; it all depends on the nature of your LINQ queries and whether the tasks they're performing lend themselves to parallelization.

To understand how PLINQ works and the situations it is useful in, it helps to see some examples. The exercises in the following sections demonstrate a pair of simple scenarios.

Using PLINQ to improve performance while iterating through a collection

The first scenario is simple. Consider a LINQ query that iterates through a collection and retrieves elements from the collection based on a processor-intensive calculation. This form of query can benefit from parallel execution as long as the calculations are independent. The elements in the collection can be divided into some partitions; the exact number depends on the current load of the computer and the number of CPUs available. The elements in each partition can be processed by a separate thread. When all the partitions have been processed, the results can be merged. Any collection that supports access to elements through an index, such as an array or a collection that implements the `ICollection<T>` interface, can be managed in this way.

To parallelize a LINQ query over a simple collection

1. Using Visual Studio 2022, open the PLINQ solution, which is located in the `\Microsoft Press\VC SBS\Chapter 24\PLINQ` folder in your **Documents** folder.

2. In Solution Explorer, double-click **Program.cs** in the PLINQ project to display the file in the Code and Text Editor window.

This is a console application. The skeleton structure of the application has been created for you. The `Program` class contains methods named `Test1` and `Test2` that illustrate a pair of common scenarios. The `Main` method calls each of these test methods in turn.

Both test methods have the same general structure: They create a LINQ query (you will add the code to do this later in this set of exercises), run it, and display the time taken. The code for each of these methods is almost completely separate from the statements that actually create and run the queries.

3. Add the following `using` directive to the list at the top of the file:

```
using System.Linq;
```

The operators used by LINQ are implemented as extension methods in this namespace.

4. Examine the `Test1` method. This method creates a large array of integers and populates it with a set of random numbers between 0 and 200. The random number generator is seeded, so you should get the same results every time you run the application.
5. Immediately after the first `// TO DO:` comment in this method, add the LINQ query shown here in bold:

```
// TO DO: Create a LINQ query that retrieves all numbers greater than 100
var over100 = from n in numbers
              where TestIfTrue(n > 100)
              select n;
```

This LINQ query retrieves all the items in the `numbers` array that have a value greater than 100. The test `n > 100` is not by itself computationally intensive enough to show the benefits of parallelizing this query, so the code calls a method named `TestIfTrue`, which slows it down a little by performing a `SpinWait` operation. The `SpinWait` method causes the processor to continually execute a loop of special “no operation” instructions for a short period, keeping the processor busy but not actually doing any work. (This is known as *spinning*.) The `TestIfTrue` method looks like this:

```
public static bool TestIfTrue(bool expr)
{
    Thread.SpinWait(100);
    return expr;
}
```

6. After the second `// TO DO:` comment in the `Test1` method, add the following code shown in bold:

```
// TO DO: Run the LINQ query, and save the results in a List<int> object
List<int> numbersOver100 = new List<int>(over100);
```

Remember that LINQ queries use deferred execution, so they don’t run until you retrieve the results from them. This statement creates a `List<int>` object and populates it with the results of running the `over100` query.

7. After the third `// TO DO:` comment in the `Test1` method, add the following statement shown in bold:

```
// TO DO: Display the results
Console.WriteLine($"There are {numbersOver100.Count} numbers over 100");
```

8. On the **Debug** menu, select **Start Without Debugging**. Note the time it takes to run `Test 1` and the number of items in the array that are greater than 100.
9. Run the application several times and calculate an average of the time. Verify that the number of items greater than 100 is the same each time. (The application uses the same random number seed each time it runs to ensure the repeatability of the tests.) Return to Visual Studio when you've finished.
10. The logic that selects each item returned by the LINQ query is independent of the selection logic for all the other items, so this query is an ideal candidate for partitioning. Modify the statement that defines the LINQ query, and specify the `AsParallel` extension method to the `numbers` array, as shown here in bold:

```
var over100 = from n in numbers.AsParallel()
              where TestIfTrue(n > 100)
              select n;
```



Note If the selection logic or calculations require access to shared data, you must synchronize the tasks that run in parallel. Otherwise, the results might be unpredictable. However, synchronization can impose an overhead and might negate the benefits of parallelizing the query.

11. On the **Debug** menu, select **Start Without Debugging**. Verify that the number of items reported by `Test1` is the same as before but that the time taken to perform the test has decreased significantly. Run the test several times and take an average of the duration required for the test.

If you're running on a dual-core processor or a twin-processor computer, you should see the time reduced by 40 to 45 percent. If you have more processor cores, the decrease should be even more dramatic. On my quad-core machine, the processing time dropped from 9.5 seconds to 3.3 seconds.

12. Close the application and return to Visual Studio.

The preceding exercise shows the performance improvement you can attain by making a small change to a LINQ query. However, keep in mind that you would see results such as this only if the calculations performed by the query take some time. I cheated a little by spinning the processor. Without this overhead, the parallel version of the query is actually slower than the serial version. In the next exercise, you'll see a LINQ query that joins two arrays in memory. This time, the exercise uses more realistic data volumes, so there's no need to slow down the query artificially.

To parallelize a LINQ query that joins two collections

1. In Solution Explorer, open the Data.cs file in the Code and Text Editor window and locate the CustomersInMemory class.

This class contains a public string array called Customers. Each string in the Customers array holds the data for a single customer, with the fields separated by commas. This format is typical of data that an application might read in from a text file that uses comma-separated fields. The first field contains the customer ID, the second field contains the name of the company that the customer represents, and the remaining fields hold the address, city, country or region, and postal code.

2. Find the OrdersInMemory class. This class is similar to the CustomersInMemory class except that it contains a string array called Orders. The first field in each string is the order number, the second field is the customer ID, and the third field is the date that the order was placed.
3. Find the OrderInfo class. This class contains four fields: customer ID, company name, order ID, and order date for an order. You'll use a LINQ query to populate a collection of OrderInfo objects from the data in the Customers and Orders arrays.
4. Display the Program.cs file in the Code and Text Editor window and locate the Test2 method in the Program class. In this method, you'll create a LINQ query that joins the Customers and Orders arrays by using the customer ID to return a list of customers and the orders that each customer has placed for all orders placed in 1997. (This is antique data!) The query will store each row of the result in an OrderInfo object.
5. In the try block in this method, after the first `// TO DO:` comment, add the code shown next in bold:

```
// TO DO: Create a LINQ query that retrieves customers and orders from arrays
// Store each row returned in an OrderInfo object
var orderInfoQuery =
    from c in CustomersInMemory.Customers.AsParallel()
    join o in OrdersInMemory.Orders.AsParallel()
    on c.Split(',')[0] equals o.Split(',')[1]
    where Convert.ToDateTime(o.Split(',')[2], new CultureInfo("en-US"))
        >= new DateTime(1997, 1, 1)
        && Convert.ToDateTime(o.Split(',')[2], new CultureInfo("en-US"))
        < new DateTime(1998, 1, 1)
    select new OrderInfo
    {
        CustomerID = c.Split(',')[0],
        CompanyName = c.Split(',')[1],
        OrderID = Convert.ToInt32(o.Split(',')[0]),
        OrderDate = Convert.ToDateTime(o.Split(',')[2],
            new CultureInfo("en-US"))
    };
};
```

This statement defines the LINQ query. Notice that it uses the `Split` method of the `String` class to split each string into an array of strings. The strings are split on the comma character. (The commas are stripped out.)

One complication is that the dates in the array are held in United States English format, so the code that converts them into `DateTime` objects specifies the United States English formatter. If you use the default formatter for your locale, the dates might not parse correctly. All in all, this query performs a significant amount of work to generate the data for each item.

6. In the `Test2` method, after the second `// TO DO:` statement, add the following code shown in bold:

```
// TO DO: Run the LINQ query, and save the results in a List<OrderInfo> object
List<OrderInfo> orderInfo = new List<OrderInfo>(orderInfoQuery);
```

This statement runs the query and populates the `orderInfo` collection.

7. After the third `// TO DO:` statement, add the statement shown here in bold:

```
// TO DO: Display the results
Console.WriteLine($"There are {orderInfo.Count} orders");
```

8. In the `Main` method, comment out the statement that calls the `Test1` method and uncomment the statement that calls the `Test2` method, as shown in the following code in bold:

```
static void Main(string[] args)
{
    // Test1();
    Test2();
}
```

9. On the **Debug** menu, select **Start Without Debugging**.
10. Verify that `Test2` retrieves 3,672 orders and note the duration of the test.
11. Run the application several times to obtain an average duration, and then return to Visual Studio.
12. In the `Test2` method, modify the LINQ query to add the `AsParallel` extension method to the `Customers` and `Orders` arrays, as shown here in bold:

```
var orderInfoQuery =
    from c in CustomersInMemory.Customers.AsParallel()
    join o in OrdersInMemory.Orders.AsParallel()
    on c.Split(',')[0] equals o.Split(',')[1]
    where Convert.ToDateTime(o.Split(',')[2], new CultureInfo("en-US"))
        >= new DateTime(1997, 1, 1)
    && Convert.ToDateTime(o.Split(',')[2], new CultureInfo("en-US"))
        < new DateTime(1998, 1, 1)
    select new OrderInfo
    {
        CustomerID = c.Split(',')[0],
        CompanyName = c.Split(',')[1],
        OrderID = Convert.ToInt32(o.Split(',')[0]),
        OrderDate = Convert.ToDateTime(o.Split(',')[2],
            new CultureInfo("en-US"))
    };
```



Warning When you join two data sources in this way, they must both be `IEnumerable` objects or `ParallelQuery` objects. This means that if you specify the `AsParallel` method for one source, you should also specify `AsParallel` for the other. If you fail to do this, your code will not run; it will stop with an error.

13. Run the application several times. Notice that the time taken for `Test2` should be significantly less than it was previously. On my system, the time dropped from an average of 440 ms to 250 ms. PLINQ can use multiple threads to optimize join operations by fetching the data for each part of the join in parallel.
14. Close the application and return to Visual Studio.

These two simple exercises have shown you the power of the `AsParallel` extension method and PLINQ. Note that PLINQ is an evolving technology, and the internal implementation is very likely to change over time. Additionally, the volumes of data and the amount of processing you perform in a query affect the usefulness of PLINQ. With small amounts of data, PLINQ can actually impose a significant overhead. Therefore, you should not regard these exercises as defining fixed rules that you should always follow. Rather, they illustrate the point that you should carefully measure and assess the likely performance or other benefits of using PLINQ with your own data, in your own environment.

Canceling a PLINQ query

Unlike with ordinary LINQ queries, you can cancel a PLINQ query. To do this, you specify a `CancellationToken` object from a `CancellationTokenSource` object and use the `WithCancellation` extension method of the `ParallelQuery`.

```
CancellationToken tok = ...;
...
var orderInfoQuery =
    from c in
        CustomersInMemory.Customers.AsParallel().WithCancellation(tok)
    join o in OrdersInMemory.Orders.AsParallel()
    on ...
```

You specify `WithCancellation` only once in a query. Cancellation applies to all sources in the query. If the `CancellationTokenSource` object used to generate the `CancellationToken` is canceled, the query stops with an `OperationCanceledException` exception.

Synchronizing concurrent access to data

PLINQ is not always the most appropriate technology to use for an application. If you create your own tasks manually, you need to ensure that these tasks coordinate their activities correctly. The .NET class library provides methods with which you can wait for tasks to complete, and you can use these methods to coordinate tasks at a very coarse level. But consider what happens if two tasks attempt to access and modify the same data. If both tasks run at the same time, their overlapping operations

might corrupt the data. This situation can lead to bugs that are difficult to correct, primarily because of their unpredictability.

The Task class provides a powerful framework with which you can design and build applications that take advantage of multiple CPU cores to perform tasks in parallel. However, you need to be careful when building solutions that perform concurrent operations, especially if those operations share access to data. You have little control over how parallel operations are scheduled or even the degree of parallelism that the operating system might provide to an application constructed by using tasks. These decisions are left as runtime considerations and depend on the workload and hardware capabilities of the computer running your application.

This level of abstraction was a deliberate design decision on the part of the Microsoft development team, and it removes the need for you to understand the low-level threading and scheduling details when you build applications that require concurrent tasks. But this abstraction comes at a cost. Although it all appears to work magically, you must make some effort to understand how your code runs; otherwise, you can end up with applications that exhibit unpredictable (and erroneous) behavior, as shown in the following example. (This sample is available in the ParallelTest project in the folder containing the code for this chapter.)

```
using System;
using System.Threading;
class Program
{
    private const int NUMELEMENTS = 10;
    static void Main(string[] args)
    {
        SerialTest();
    }

    static void SerialTest()
    {
        int[] data = new int[NUMELEMENTS];
        int j = 0;
        for (int i = 0; i < NUMELEMENTS; i++)
        {
            j = i;
            doAdditionalProcessing();
            data[i] = j;
            doMoreAdditionalProcessing();
        }

        for (int i = 0; i < NUMELEMENTS; i++)
        {
            Console.WriteLine($"Element {i} has value {data[i]}");
        }
    }

    static void doAdditionalProcessing()
    {
        Thread.Sleep(10);
    }
}
```

```

        static void doMoreAdditionalProcessing()
        {
            Thread.Sleep(10);
        }
    }

```

The `SerialTest` method populates an integer array with a set of values (in a rather long-winded way) and then iterates through this list, printing the index of each item in the array together with the value of the corresponding item. The `doAdditionalProcessing` and `doMoreAdditionalProcessing` methods simply simulate the performance of long-running operations as part of the processing that might cause the runtime to yield control of the processor. The output of the program method is shown here:

```

Element 0 has value 0
Element 1 has value 1
Element 2 has value 2
Element 3 has value 3
Element 4 has value 4
Element 5 has value 5
Element 6 has value 6
Element 7 has value 7
Element 8 has value 8
Element 9 has value 9

```

Now consider the `ParallelTest` method, shown next. This method is the same as the `SerialTest` method except that it uses the `Parallel.For` construct to populate the data array by running concurrent tasks. The code in the lambda expression run by each task is identical to that in the initial `for` loop in the `SerialTest` method.

```

using System.Threading.Tasks;
...
static void ParallelTest()
{
    int[] data = new int[NUMELEMENTS];
    int j = 0;
    Parallel.For (0, NUMELEMENTS, (i) =>
    {
        j = i;
        doAdditionalProcessing();
        data[i] = j;
        doMoreAdditionalProcessing();
    });

    for (int i = 0; i < NUMELEMENTS; i++)
    {
        Console.WriteLine($"Element {i} has value {data[i]}");
    }
}

```

The intention is for the `ParallelTest` method to perform the same operation as the `SerialTest` method, except by using concurrent tasks and (with good luck) running a little faster as a result. The problem is that it might not always work as expected. Some sample output generated by the `ParallelTest` method is shown here:

```
Element 0 has value 2
Element 1 has value 1
Element 2 has value 6
Element 3 has value 1
Element 4 has value 8
Element 5 has value 9
Element 6 has value 8
Element 7 has value 9
Element 8 has value 8
Element 9 has value 9
```

The values assigned to each item in the data array are not always the same as the values generated by using the `SerialTest` method. Additionally, further runs of the `ParallelTest` method can produce different sets of results.

If you examine the logic in the `Parallel.For` construct, you should see where the problem lies. The lambda expression contains the following statements:

```
j = i;
doAdditionalProcessing();
data[i] = j;
doMoreAdditionalProcessing();
```

The code looks innocuous enough. It copies the current value of the variable *i* (the index variable identifying which iteration of the loop is running) into the variable *j*, and later it stores the value of *j* in the element of the data array indexed by *i*. If *i* contains 5, *j* is assigned the value 5, and later the value of *j* is stored in `data[5]`. But between assigning the value to *j* and then reading it back, the code does more work: it calls the `doAdditionalProcessing` method. If this method takes a long time to execute, the runtime might suspend the thread and schedule another task. A concurrent task that's running another iteration of the `Parallel.For` construct might run and assign a new value to *j*. Consequently, when the original task resumes, the value of *j* it assigns to `data[5]` is not the value it stored, and the result is data corruption. More troublesome is that sometimes this code might run as expected and produce the correct results, and at other times it might not; it all depends on how busy the computer is and when the various tasks are scheduled. Consequently, these types of bugs can lie dormant during testing and then suddenly manifest in a production environment.

The variable *j* is shared by all the concurrent tasks. If a task stores a value in *j* and later reads it back, it must ensure that no other task has modified *j* in the meantime. This requires synchronizing access to the variable across all concurrent tasks that can access it. One way you can achieve synchronized access is to lock data.

Locking data

The C# language provides locking semantics through the `lock` keyword, which you can use to guarantee exclusive access to resources. You use the `lock` keyword like this:

```
object myLockObject = new object();  
...  
lock (myLockObject)  
{  
    // Code that requires exclusive access to a shared resource  
    ...  
}
```

The `lock` statement attempts to obtain a mutual-exclusion lock over the specified object (you can actually use any reference type, not just `object`), and it blocks execution if this same object is currently locked by another thread. When the thread obtains the lock, the code in the block following the `lock` statement runs. At the end of this block, the lock is released. If another thread is blocked waiting for the lock, it can then grab the lock and continue its processing.

Synchronization primitives for coordinating tasks

The `lock` keyword is fine for many simple scenarios, but in some situations, you might have more complex requirements. The `System.Threading` namespace includes some additional synchronization primitives that you can use to address these situations. These synchronization primitives are classes designed for use with tasks; they expose locking mechanisms that restrict access to a resource while a task holds the lock.

These primitives support a variety of locking techniques that you can use to implement different styles of concurrent access, ranging from simple exclusive locks (where a single task has sole access to a resource), to semaphores (where multiple tasks can access a resource simultaneously but in a controlled manner), to reader/writer locks (which enable different tasks to share read-only access to a resource while guaranteeing exclusive access to a thread that needs to modify the resource). The following sections summarize some of these primitives. For more information and examples, consult the documentation provided with Visual Studio 2022.



Note The .NET Framework has included a respectable set of synchronization primitives since its initial release. The following list describes only the more recent primitives included in the `System.Threading` namespace. There is some overlap between the new primitives and those provided previously. Where overlapping functionality exists, you should use the more recent alternatives because they've been designed and optimized for computers with multiple CPUs.

A detailed discussion of the theory of all the possible synchronization mechanisms available for building multithreaded applications is beyond the scope of this book. For more information about the general theory of multiple threads and synchronization, see the topic “Synchronizing Data for Multithreading” in the documentation provided with Visual Studio 2022.

ManualResetEventSlim

The `ManualResetEventSlim` class provides functionality by which one or more tasks can wait for an event. A `ManualResetEventSlim` object can be in one of two states: `signaled (true)` and `unsignaled (false)`. A task creates a `ManualResetEventSlim` object and specifies its initial state. Other tasks can wait for the `ManualResetEventSlim` object to be signaled by calling the `Wait` method. If the `ManualResetEventSlim` object is in the `unsignaled` state, the `Wait` method blocks the tasks. Another task can change the state of the `ManualResetEventSlim` object to `signaled` by calling the `Set` method. This action releases all tasks waiting on the `ManualResetEventSlim` object, which can then resume running. The `Reset` method changes the state of a `ManualResetEventSlim` object back to `unsignaled`.

SemaphoreSlim

You can use the `SemaphoreSlim` class to control access to a pool of resources. A `SemaphoreSlim` object has an initial value (a nonnegative integer) and an optional maximum value. Typically, the initial value of a `SemaphoreSlim` object is the number of resources in the pool. Tasks accessing the resources in the pool first call the `Wait` method. This method attempts to decrement the value of the `SemaphoreSlim` object. If the result is nonzero, the thread is allowed to continue and can take a resource from the pool. When it has finished, the task should call the `Release` method on the `SemaphoreSlim` object. This action increments the value of the `Semaphore`.

If a task calls the `Wait` method and the result of decrementing the value of the `SemaphoreSlim` object would result in a negative value, the task waits until another task calls `Release`.

The `SemaphoreSlim` class also provides the `CurrentCount` property, which you can use to determine whether a `Wait` operation is likely to succeed immediately or will result in blocking.

CountdownEvent

You can think of the `CountdownEvent` class as a cross between the inverse of a semaphore and a manual reset event. When a task creates a `CountdownEvent` object, it specifies an initial value (a non-negative integer). One or more tasks can call the `Wait` method of the `CountdownEvent` object, and if its value is nonzero, the tasks are blocked. `Wait` does not decrement the value of the `CountdownEvent` object; instead, other tasks can call the `Signal` method to reduce the value. When the value of the `CountdownEvent` object reaches 0, all blocked tasks are signaled and can resume running.

A task can set the value of a `CountdownEvent` object back to the value specified in its constructor by using the `Reset` method, and a task can increase this value by calling the `AddCount` method. You can determine whether a call to `Wait` is likely to block by examining the `CurrentCount` property.

ReaderWriterLockSlim

The `ReaderWriterLockSlim` class is an advanced synchronization primitive that supports a single writer and multiple readers. The idea is that modifying (writing to) a resource requires exclusive access, but reading a resource does not. Multiple readers can access the same resource at the same time, but not at the same time as a writer.

A task that wants to read a resource calls the `EnterReadLock` method of a `ReaderWriterLockSlim` object. This action grabs a read lock on the object. When the task has finished with the resource, it calls the `ExitReadLock` method, which releases the read lock. Multiple tasks can read the same resource at the same time, and each task obtains its own read lock.

When a task modifies the resource, it can call the `EnterWriteLock` method of the same `ReaderWriterLockSlim` object to obtain a write lock. If one or more tasks currently have a read lock for this object, the `EnterWriteLock` method blocks until they're all released. After a task has a write lock, it can then modify the resource and call the `ExitWriteLock` method to release the lock.

A `ReaderWriterLockSlim` object has only a single write lock. If another task attempts to obtain the write lock, it is blocked until the first task releases this write lock. To ensure that writing tasks are not blocked indefinitely, as soon as a task requests the write lock, all subsequent calls to `EnterReadLock` made by other tasks are blocked until the write lock has been obtained and released.

Barrier

With the `Barrier` class, you can temporarily halt the execution of a set of tasks at a particular point in an application and continue only when all tasks have reached this point. It's useful for synchronizing tasks that need to perform a series of concurrent operations in step with one another.

When a task creates a `Barrier` object, it specifies the number of tasks in the set that will be synchronized. You can think of this value as a task counter maintained internally inside the `Barrier` class. This value can be amended later by calling the `AddParticipant` or `RemoveParticipant` method. When a task reaches a synchronization point, it calls the `SignalAndWait` method of the `Barrier` object, which decrements the thread counter inside the `Barrier` object. If this counter is greater than 0, the task is blocked. Only when the counter reaches 0 are all the tasks waiting on the `Barrier` object released, and only then can they continue running.

The `Barrier` class provides the `ParticipantCount` property, which specifies the number of tasks that it synchronizes, and the `ParticipantsRemaining` property, which indicates how many tasks need to call `SignalAndWait` before the barrier is raised and blocked tasks can continue running.

You can also specify a delegate in the `Barrier` constructor. This delegate can refer to a method that runs when all the tasks have arrived at the barrier. The `Barrier` object is passed in as a parameter to this method. The barrier is not raised, and the tasks are not released until this method completes.

Canceling synchronization

The `ManualResetEventSlim`, `SemaphoreSlim`, `CountdownEvent`, and `Barrier` classes all support cancellation by following the cancellation model described in Chapter 23. The wait operations for each of these classes can take an optional `CancellationToken` parameter, retrieved from a `CancellationTokenSource` object. If you call the `Cancel` method of the `CancellationTokenSource` object, each wait operation referencing a `CancellationToken` generated from this source is aborted with an `OperationCanceledException` exception (possibly wrapped in an `AggregateException` exception, depending on the context of the wait operation).

The following code shows how to invoke the `Wait` method of a `SemaphoreSlim` object and specify a cancellation token. If the wait operation is canceled, the `OperationCanceledException` catch handler runs.

```
CancellationTokenSource = new CancellationTokenSource();
CancellationToken = cancellationTokenSource.Token;
...
// Semaphore that protects a pool of 3 resources
SemaphoreSlim semaphoreSlim = new SemaphoreSlim(3);
...
// Wait on the semaphore, and catch the OperationCanceledException if
// another thread calls Cancel on cancellationTokenSource
try
{
    semaphoreSlim.Wait(cancellationToken);
}
catch (OperationCanceledException e)
{
    ...
}
```

The concurrent collection classes

A common requirement of many multithreaded applications is to store and retrieve data in a collection. The standard collection classes provided with .NET are not thread-safe by default, although you can use the synchronization primitives described previously to wrap code that adds, queries, and removes elements in a collection. However, this process is potentially prone to error and isn't very scalable, so the .NET class library includes a small set of thread-safe collection classes and interfaces in the `System.Collections.Concurrent` namespace that are designed specifically for use with tasks. The following list briefly summarizes the key types in this namespace:

- **`ConcurrentBag<T>`** This is a general-purpose class for holding an unordered collection of items. It includes methods to insert (`Add`), remove (`TryTake`), and examine (`TryPeek`) items in the collection. These methods are thread-safe. The collection is also enumerable, so you can iterate over its contents by using a `foreach` statement.
- **`ConcurrentDictionary<TKey, TValue>`** This class implements a thread-safe version of the generic `Dictionary<TKey, TValue>` collection class described in Chapter 18, "Using collections." It provides the methods `TryAdd`, `ContainsKey`, `TryGetValue`, `TryRemove`, and `TryUpdate`, which you can use to add, query, remove, and modify items in the dictionary.
- **`ConcurrentQueue<T>`** This class provides a thread-safe version of the generic `Queue<T>` class described in Chapter 18. It includes the methods `Enqueue`, `TryDequeue`, and `TryPeek`, which you can use to add, remove, and query items in the queue.
- **`ConcurrentStack<T>`** This is a thread-safe implementation of the generic `Stack<T>` class, also described in Chapter 18. It provides methods such as `Push`, `TryPop`, and `TryPeek`, which you can use to push, pop, and query items on the stack.

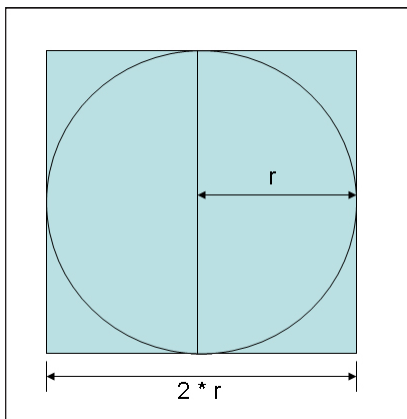


Note Adding thread safety to the methods in a collection class imposes additional runtime overhead, so these classes are not as fast as the regular collection classes. Keep this in mind when deciding whether to parallelize a set of operations that require access to a shared collection.

Using a concurrent collection and a lock to implement thread-safe data access

In the following set of exercises, you'll implement an application that calculates pi by using a geometric approximation. Initially, you'll perform the calculation in a single-threaded manner; then you'll change the code to perform the calculation by using parallel tasks. In the process, you will uncover some data-synchronization issues, which you'll solve by using a concurrent collection class and a lock to ensure that the tasks coordinate their activities correctly.

The algorithm that you'll implement calculates pi based on some simple mathematics and statistical sampling. If you draw a circle of radius r and draw a square with sides that touch the circle, the sides of the square are $2 * r$ in length, as shown in the following image:



You can calculate the area of the square, S , like this:

$$S = (2 * r) * (2 * r)$$

or

$$S = 4 * r * r$$

The area of the circle, C , is calculated as follows:

$$C = \pi * r * r$$

Rearranging these formulas, you can see that

$$r * r = C / \pi$$

and

$$r * r = S / 4$$

Combining these equations, you get:

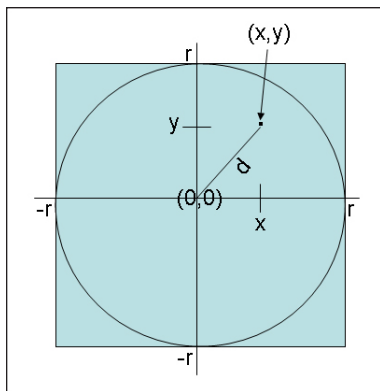
$$S / 4 = C / \pi$$

And therefore:

$$\pi = 4 * C / S$$

The trick is to determine the value of the ratio of the area of the circle, C , with respect to the area of the square, S . This is where the statistical sampling comes in. You can generate a set of random points that lie within the square and count how many of these points also fall within the circle. If you generate a sufficiently large and random sample, the ratio of points that lie within the circle to the points that lie within the square (and also in the circle) approximates the ratio of the areas of the two shapes, C / S . All you have to do is count them.

How do you determine whether a point lies within the circle? To help visualize the solution, draw the square on a piece of graph paper with the center of the square at the origin, point $(0,0)$. You can then generate pairs of values, or coordinates, that lie within the range $(-r, -r)$ to $(+r, +r)$. You can determine whether any coordinates (x, y) lie within the circle by applying Pythagoras' theorem to determine the distance d of these coordinates from the origin. You can calculate d as the square root of $((x * x) + (y * y))$. If d is less than or equal to r , the radius of the circle, the coordinates (x, y) specify a point within the circle, as shown in the following diagram:



You can simplify matters further by generating coordinates that lie only in the upper-right quadrant of the graph so that you only have to generate pairs of random numbers between 0 and r . This is the approach you will take in the exercises.



Note The following exercises are intended to be run on a computer with a multicore processor. If you have only a single-core CPU, you won't observe the same effects. Also, you should not start any additional programs or services between exercises, because these might affect the results you see.

To calculate pi by using a single thread

1. Start Visual Studio 2022, if it is not already running.
2. Open the **CalculatePI** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 24\CalculatePI** folder in your **Documents** folder.
3. In Solution Explorer, in the CalculatePI project, double-click **Program.cs** to display the file in the Code and Text Editor window. This is a console application. The skeleton structure of the application has already been created for you.
4. Scroll to the bottom of the file and examine the `Main` method. It looks like this:

```
static void Main(string[] args)
{
    double pi = SerialPI();
    Console.WriteLine($"Geometric approximation of PI calculated serially: {pi}");

    Console.WriteLine();

    // pi = ParallelPI();
    // Console.WriteLine($"Geometric approximation of PI calculated in parallel: {pi}");
}
```

This code calls the `SerialPI` method, which calculates pi by using the geometric algorithm described before this exercise. The value is returned as a `double` and displayed. The code that's currently commented out calls the `ParallelPI` method, which performs the same calculation but by using concurrent tasks. The result displayed should be the same as that returned by the `SerialPI` method.

5. Examine the `SerialPI` method.

```
static double SerialPI()
{
    List<double> pointsList = new List<double>();
    Random random = new Random(SEED);
    int numPointsInCircle = 0;
    Stopwatch timer = new Stopwatch();
    timer.Start();
    try
    {
        // TO DO: Implement the geometric approximation of PI
        return 0;
    }
    finally
    {
        long milliseconds = timer.ElapsedMilliseconds;
        Console.WriteLine($"SerialPI complete: Duration: {milliseconds} ms");
        Console.WriteLine(
            $"Points in pointsList: {pointsList.Count}. Points within circle: {numPointsInCircle}");
    }
}
```

This method generates a large set of coordinates and calculates the distances of each set of coordinates from the origin. The size of the set is specified by the constant `NUMPOINTS` at the top of the `Program` class. The bigger this value is, the greater the set of coordinates, and the more accurate the value of `pi` calculated by this method. If your computer has sufficient memory, you can increase the value of `NUMPOINTS`. Similarly, if you find that the application throws `OutOfMemoryException` exceptions when you run it, you can reduce this value.

You store the distance of each point from the origin in the `pointsList` `List<double>` collection. The data for the coordinates is generated by using the `random` variable. This is a `Random` object, seeded with a constant to generate the same set of random numbers each time you run the program. (This helps you determine that it's running correctly.) You can change the `SEED` constant at the top of the `Program` class if you want to seed the random number generator with a different value.

You use the `numPointsInCircle` variable to count the number of points in the `pointsList` collection that lie within the bounds of the circle. The radius of the circle is specified by the `RADIUS` constant at the top of the `Program` class.

To help you compare performance between this method and the `ParallelPI` method, the code creates a `Stopwatch` variable called `timer` and starts it running. The `finally` block determines how long the calculation took and displays the result. For reasons that will be described later, the `finally` block also displays the number of items in the `pointsList` collection and the number of points that it found that lie within the circle.

You'll add the code that actually performs the calculation to the `try` block in the next few steps.

6. In the `try` block, delete the comment and remove the `return` statement. (This statement was provided only to ensure that the code compiles.) Then add to the `try` block the `for` block and statements shown in bold in the following code:

```
try
{
    for (int points = 0; points < NUMPOINTS; points++)
    {
        int xCoord = random.Next(RADIUS);
        int yCoord = random.Next(RADIUS);
        double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
        pointsList.Add(distanceFromOrigin);
        doAdditionalProcessing();
    }
}
```

This block of code generates a pair of coordinate values that lie in the range 0 to `RADIUS`, and it stores them in the `xCoord` and `yCoord` variables. The code then employs Pythagoras' theorem to calculate the distance of these coordinates from the origin and adds the result to the `pointsList` collection.



Note Although this code performs a little bit of computational work, in a real-world scientific application, you're likely to include far more complex calculations that will keep the processor occupied for longer. To simulate this situation, this code calls another method, `doAdditionalProcessing`. All this method does is occupy a number of CPU cycles, as shown in the following code sample. I opted to follow this approach to better demonstrate the data-synchronization requirements of multiple tasks rather than have you write an application that performs a highly complex calculation such as a fast Fourier transform (FFT) to keep the CPU busy.

```
private static void doAdditionalProcessing()
{
    Thread.SpinWait(SPINWAIT);
}
```

`SPINWAIT` is another constant defined at the top of the `Program` class.

7. In the `SerialPI` method, in the `try` block, after the `for` block, add the `foreach` statement shown in bold in the following example:

```
try
{
    for (int points = 0; points < NUMPOINTS; points++)
    {
        ...
    }

    foreach (double datum in pointsList)
    {
        if (datum <= RADIUS)
        {
            numPointsInCircle++;
        }
    }
}
```

This code iterates through the `pointsList` collection and examines each value in turn. If the value is less than or equal to the radius of the circle, it increments the `numPointsInCircle` variable. At the end of this loop, `numPointsInCircle` should contain the total number of coordinates that were found to lie within the bounds of the circle.

8. After the `foreach` statement, add to the `try` block the following statements shown in bold:

```
try
{
    for (int points = 0; points < NUMPOINTS; points++)
    {
        ...
    }

    foreach (double datum in pointsList)
    {
        ...
    }
}
```

```

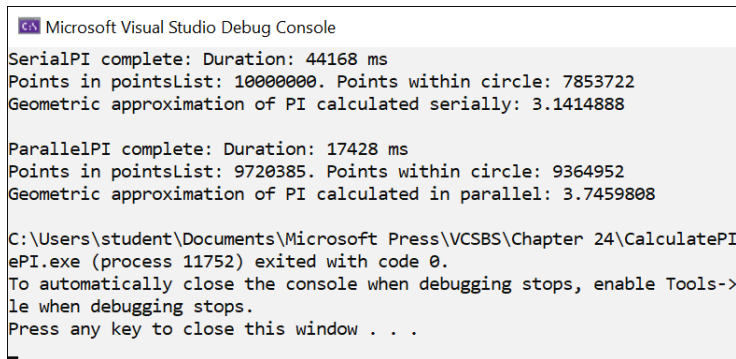
        double pi = 4.0 * numPointsInCircle / NUMPOINTS;
        return pi;
    }

```

The first statement calculates pi based on the ratio of the number of points that lie within the circle to the total number of points, using the formula described earlier. The value is returned as the result of the method.

9. On the **Debug** menu, select **Start Without Debugging**.

The program runs and displays its approximation of pi, as shown in the following image. (It took over 44 seconds on my computer, so be prepared to wait for a little while.) The time taken to calculate the result also appears.



```

Microsoft Visual Studio Debug Console

SerialPI complete: Duration: 44168 ms
Points in pointsList: 10000000. Points within circle: 7853722
Geometric approximation of PI calculated serially: 3.1414888

ParallelPI complete: Duration: 17428 ms
Points in pointsList: 9720385. Points within circle: 9364952
Geometric approximation of PI calculated in parallel: 3.7459808

C:\Users\student\Documents\Microsoft Press\VCSBS\Chapter 24\CalculatePI
ePI.exe (process 11752) exited with code 0.
To automatically close the console when debugging stops, enable Tools->
le when debugging stops.
Press any key to close this window . . .

```



Note Apart from the timing, your result should be the same as that shown, 3.1414888, unless you've changed the NUMPOINTS, RADIUS, or SEED constants.

10. Close the console window and return to Visual Studio.

In the SerialPI method, the code in the for loop that generates the points and calculates their distance from the origin is an obvious area that can be parallelized. This is what you'll do in the next exercise.

To calculate pi by using parallel tasks

1. In Solution Explorer, double-click **Program.cs** to display the file in the Code and Text Editor window if it is not already open.
2. Locate the ParallelPI method. It contains the same code as the initial version of the SerialPI method before you added the code to the try block to calculate pi.
3. In the try block, delete the comment and remove the return statement. Then add the Parallel.For statement shown here in bold to the try block:

```

try
{
    Parallel.For (0, NUMPOINTS, (x) =>

```

```

    {
        int xCoord = random.Next(RADIUS);
        int yCoord = random.Next(RADIUS);
        double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
        pointsList.Add(distanceFromOrigin);
        doAdditionalProcessing();
    });
}

```

This construct is the parallel analog of the code in the for loop in the `SerialPI` method. The body of the original for loop is wrapped in a lambda expression. Remember that each iteration of the loop is performed by using a task, and tasks can run in parallel. The degree of parallelism depends on the number of processor cores and other resources available on your computer.

4. Add the following code shown in bold to the try block, after the `Parallel.For` statement. This code is the same as the corresponding statements in the `SerialPI` method.

```

try
{
    Parallel.For (...
    {
        ...
    });

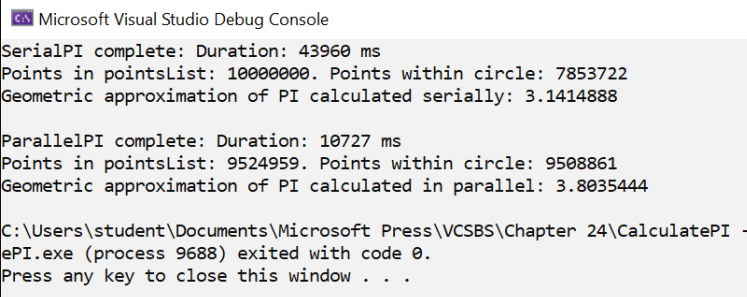
    foreach (double datum in pointsList)
    {
        if (datum <= RADIUS)
        {
            numPointsInCircle++;
        }
    }

    double pi = 4.0 * numPointsInCircle / NUMPOINTS;
    return pi;
}

```

5. In the `Main` method near the end of the `Program.cs` file, uncomment the code that calls the `ParallelPI` method and the `Console.WriteLine` statement that displays the results.
6. On the **Debug** menu, select **Start Without Debugging**.

The program runs. The following image shows the typical output. (Your timings might be different; I was using a quad-core processor.)



```

Microsoft Visual Studio Debug Console

SerialPI complete: Duration: 43960 ms
Points in pointsList: 10000000. Points within circle: 7853722
Geometric approximation of PI calculated serially: 3.1414888

ParallelPI complete: Duration: 10727 ms
Points in pointsList: 9524959. Points within circle: 9508861
Geometric approximation of PI calculated in parallel: 3.8035444

C:\Users\student\Documents\Microsoft Press\VCSBS\Chapter 24\CalculatePI -
ePI.exe (process 9688) exited with code 0.
Press any key to close this window . . .

```


The `ParallelPi` method runs a lot more quickly than the `SerialPi` method. However, although the value calculated by the `SerialPi` method should be exactly as before, the result of the `ParallelPi` method, 3.8035444, looks a little suspect. The random number generator is seeded with the same value as that used by the `SerialPi` method, so it should produce the same sequence of random numbers with the same result and the same number of points within the circle. Moreover, if you run the application again, you should get the same value of pi for the `SerialPi` method, but the value calculated by the `ParallelPi` method is likely to be different (and still inaccurate). Another curious point is that the `pointsList` collection in the `ParallelPi` method seems to contain fewer points than the same collection in the `SerialPi` method.



Note If the `pointsList` collection actually contains the expected number of items, run the application again. You should find that it contains fewer items than expected in most (but not necessarily all) runs.

7. Close the console window and return to Visual Studio.

What went wrong with the parallel calculation? A good place to start is the number of items in the `pointsList` collection. This collection is a generic `List<double>` object. However, this type is not thread-safe. The code in the `Parallel` .For statement calls the `Add` method to append a value to the collection, but remember that this code is being executed by tasks running as concurrent threads. Consequently, given the number of items being added to the collection, it's highly probable that some of the calls to `Add` will interfere with one another and cause some corruption. A solution is to use a collection from the `System.Collections.Concurrent` namespace because these collections are thread-safe. The generic `ConcurrentBag<T>` class in this namespace is probably the most suitable collection to use for this example.

To use a thread-safe collection

1. In Solution Explorer, double-click **Program.cs** to display the file in the Code and Text Editor window if it is not already open.
2. Add the following `using` directive to the list at the top of the file:
`using System.Collections.Concurrent;`
3. Locate the `ParallelPi` method. At the start of this method, replace the statement that instantiates the `List<double>` collection with code that creates a `ConcurrentBag<double>` collection instead, as shown in bold in the following code example:

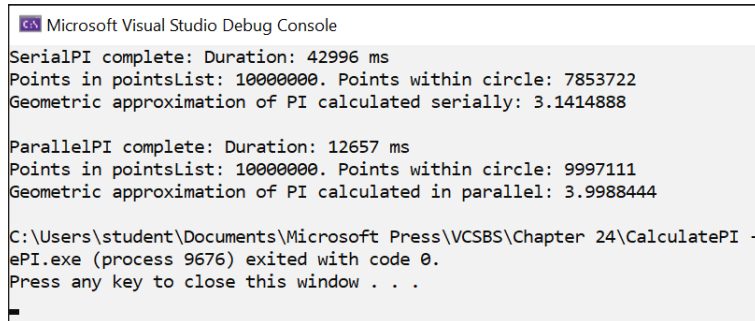
```
static double ParallelPi()
{
    ConcurrentBag<double> pointsList = new ConcurrentBag<double>();
    Random random = ...;
    ...
}
```

Notice that you cannot specify a default capacity for this class, so the constructor does not take a parameter.

You don't need to change any other code in this method; you add an item to a `ConcurrentBag<T>` collection by using the `Add` method, which is the same mechanism that you use to add an item to a `List<T>` collection.

4. On the **Debug** menu, select **Start Without Debugging**.

The program runs and displays its approximation of pi by using the `SerialPI` and `ParallelPI` methods. The following image shows the typical output:



```
Microsoft Visual Studio Debug Console
SerialPI complete: Duration: 42996 ms
Points in pointsList: 10000000. Points within circle: 7853722
Geometric approximation of PI calculated serially: 3.1414888

ParallelPI complete: Duration: 12657 ms
Points in pointsList: 10000000. Points within circle: 9997111
Geometric approximation of PI calculated in parallel: 3.9988444

C:\Users\student\Documents\Microsoft Press\VCSBS\Chapter 24\CalculatePI -
ePI.exe (process 9676) exited with code 0.
Press any key to close this window . . .
```

This time, the `pointsList` collection in the `ParallelPI` method contains the correct number of points, but the number of points within the circle, 9997111, appears to be very high. It should be the same as that reported by the `SerialPI` method.

Also note that the time taken by the `ParallelPI` method has increased compared with the previous exercise. This is because the methods in the `ConcurrentBag<T>` class have to lock and unlock data to guarantee thread safety, and this process adds to the overhead of calling these methods. Keep this in mind when you're considering whether it's appropriate to parallelize an operation.

5. Close the console window, and return to Visual Studio.

You currently have the correct number of points in the `pointsList` collection, but the value recorded for each of these points is now questionable. The code in the `Parallel` .`For construct` calls the `Next` method of a `Random` object, but like the methods in the generic `List<T>` class, this method is not thread-safe. Sadly, there is no concurrent version of the `Random` class, so you must resort to using an alternative technique to serialize calls to the `Next` method. Because each invocation is relatively brief, it makes sense to use a simple lock to guard calls to this method.

To use a lock to serialize method calls

1. In Solution Explorer, double-click **Program.cs** to display the file in the Code and Text Editor window if it is not already open.
2. Locate the `ParallelPI` method. Then modify the code in the lambda expression in the `Parallel`. For statement to protect the calls to `random`. Next by using a `lock` statement. Specify the `pointsList` collection as the subject of the lock, as shown here in bold:

```
static double ParallelPI()
{
    ...
    Parallel.For(0, NUMPOINTS, (x) =>
    {
        int xCoord;
        int yCoord;

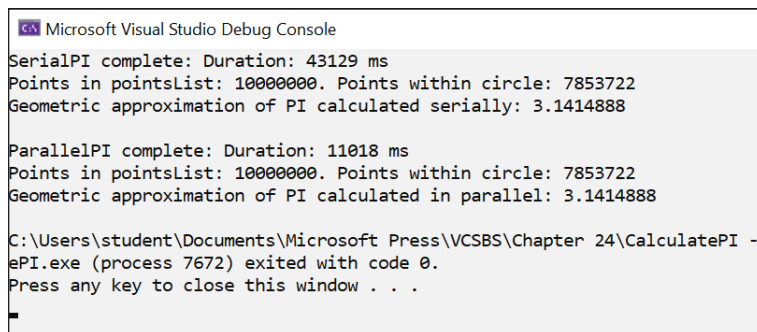
        lock(pointsList)
        {
            xCoord = random.Next(RADIUS);
            yCoord = random.Next(RADIUS);
        }

        double distanceFromOrigin = Math.Sqrt(xCoord * xCoord + yCoord * yCoord);
        pointsList.Add(distanceFromOrigin);
        doAdditionalProcessing();
    });
    ...
}
```

Notice that the `xCoord` and `yCoord` variables are declared outside the `lock` statement. You do this because the `lock` statement defines its own scope, and any variables defined within the block specifying the scope of the `lock` statement disappear when the construct exits.

3. On the **Debug** menu, select **Start Without Debugging**.

This time, the values of pi calculated by the `SerialPI` and `ParallelPI` methods are the same. The only difference is that the `ParallelPI` method runs more much more quickly.



```
Microsoft Visual Studio Debug Console
SerialPI complete: Duration: 43129 ms
Points in pointsList: 10000000. Points within circle: 7853722
Geometric approximation of PI calculated serially: 3.1414888

ParallelPI complete: Duration: 11018 ms
Points in pointsList: 10000000. Points within circle: 7853722
Geometric approximation of PI calculated in parallel: 3.1414888

C:\Users\student\Documents\Microsoft Press\VCSBS\Chapter 24\CalculatePI -
ePI.exe (process 7672) exited with code 0.
Press any key to close this window . . .
```

4. Close the console window, and return to Visual Studio.

Summary

In this chapter, you saw how to define asynchronous methods by using the `async` modifier and the `await` operator. Asynchronous methods are based on tasks, and the `await` operator specifies the points at which a task can be used to perform asynchronous processing.

You also learned a little about PLINQ and how you can use the `AsParallel` extension method to parallelize some LINQ queries. However, PLINQ is a big subject in its own right, and this chapter has only shown you how to get started. For more information, see the topic “Parallel LINQ (PLINQ)” in the documentation provided with Visual Studio.

This chapter also showed you how to synchronize data access in concurrent tasks by using the synchronization primitives provided for use with tasks. You saw how to use the concurrent collection classes to maintain collections of data in a thread-safe manner.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 25, “Implementing the user interface for a Universal Windows Platform app.”
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

To	Do this
Implement an asynchronous method	Define the method with the <code>async</code> modifier and change the type of the method to return a <code>Task</code> object (or a <code>void</code>). In the body of the method, use the <code>await</code> operator to specify points at which asynchronous processing can be performed. For example: <pre>private async Task<int> calculateValueAsync(...) { // Invoke calculateValue using a Task Task<int> generateResultTask = Task.Run(() => calculateValue(...)); await generateResultTask; return generateResultTask.Result; }</pre>
Parallelize a LINQ query	Specify the <code>AsParallel</code> extension method with the data source in the query. For example: <pre>var over100 = from n in numbers.AsParallel() where ... select n;</pre>
Enable cancellation in a PLINQ query	Use the <code>WithCancellation</code> method of the <code>ParallelQuery</code> class in the PLINQ query and specify a cancellation token. For example: <pre>CancellationToken tok = ...; ... var orderInfoQuery = from c in CustomersInMemory.Customers. AsParallel().WithCancellation(tok) join o in OrdersInMemory.Orders.AsParallel() on ...</pre>

To	Do this
Synchronize one or more tasks to implement thread-safe exclusive access to shared data	Use the <code>lock</code> statement to guarantee exclusive access to the data. For example: <pre>object myLockObject = new object(); ... lock (myLockObject) { // Code that requires exclusive access // to a shared resource ... }</pre>
Synchronize threads and make them wait for an event	Use a <code>ManualResetEventSlim</code> object to synchronize an indeterminate number of threads. Use a <code>CountdownEvent</code> object to wait for an event to be signaled a specified number of times. Use a <code>Barrier</code> object to coordinate a specified number of threads and synchronize them at a particular point in an operation.
Synchronize access to a shared pool of resources	Use a <code>SemaphoreSlim</code> object. Specify the number of items in the pool in the constructor. Call the <code>Wait</code> method prior to accessing a resource in the shared pool. Call the <code>Release</code> method when you've finished with the resource. For example: <pre>SemaphoreSlim semaphore = new SemaphoreSlim(3); ... semaphore.Wait(); // Access a resource from the pool ... semaphore.Release();</pre>
Provide exclusive write access to a resource but shared read access	Use a <code>ReaderWriterLockSlim</code> object. Before reading the shared resource, call the <code>EnterReadLock</code> method. Call the <code>ExitReadLock</code> method when you've finished. Before writing to the shared resource, call the <code>EnterWriteLock</code> method. Call the <code>ExitWriteLock</code> method when you've completed the write operation. For example: <pre>ReaderWriterLockSlim readerWriterLock = new ReaderWriterLockSlim(); Task readerTask = Task.Factory.StartNew(() => { readerWriterLock.EnterReadLock(); // Read shared resource readerWriterLock.ExitReadLock(); }); Task writerTask = Task.Factory.StartNew(() => { readerWriterLock.EnterWriteLock(); // Write to shared resource readerWriterLock.ExitWriteLock(); });</pre>
Cancel a blocking wait operation	Create a cancellation token from a <code>CancellationTokenSource</code> object and specify this token as a parameter to the wait operation. To cancel the wait operation, call the <code>Cancel</code> method of the <code>CancellationTokenSource</code> object. For example: <pre>CancellationTokenSource = new CancellationTokenSource(); CancellationToken = cancellationTokenSource.Token; ... // Semaphore that protects a pool of 3 resources SemaphoreSlim = new SemaphoreSlim(3); ... // Wait on the semaphore, and throw an // OperationCanceledException if // another thread calls Cancel on // cancellationTokenSource semaphore.Wait(cancellationToken);</pre>

Implementing the user interface for a Universal Windows Platform app

After completing this chapter, you will be able to:

- Describe the features of a typical Universal Windows Platform (UWP) app.
- Use the Blank App template as the basis for a Universal Windows Platform app.
- Implement a scalable user interface for a Universal Windows Platform app that can adapt to different form factors and device orientations.
- Create and apply styles to a Universal Windows Platform app.

Windows provides a platform for building and running highly interactive applications with continuously connected, touch-driven user interfaces and support for embedded device sensors. This platform is called the Windows Runtime (WinRT). WinRT provides a programmatic interface to the underlying Windows operating system, enabling you to build applications that can access and control Windows resources.

You can use Visual Studio to build WinRT applications that can adapt themselves to a variety of device form factors, ranging from handheld tablets to desktop PCs with large, high-resolution screens. Using Windows 11 and Visual Studio 2022, you can also publish these applications in the Microsoft Store as Microsoft Store apps.

Different devices provide a range of capabilities and graphical user interface (GUI) layout options. Microsoft developed the Universal Windows Platform (UWP), which runs on top of WinRT, as an abstraction layer that enables you to build GUI apps that can run on a variety of hardware without requiring you to modify the code. You've been building and running UWP apps throughout this book; most differences are hidden in configuration options that a user can set on their device.

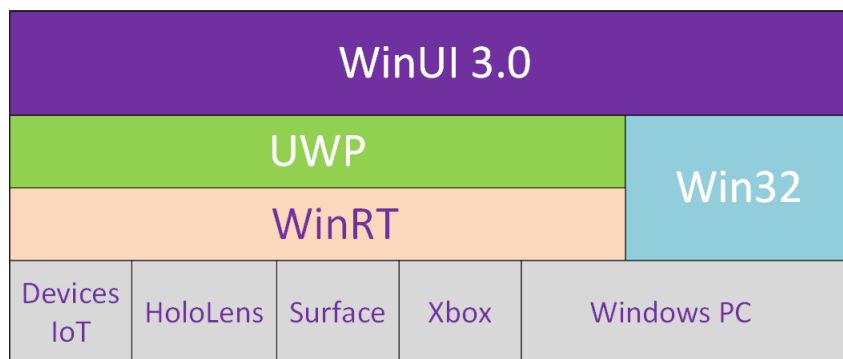
Using the UWP, you can build applications that will run on the widest range of Windows devices without the need to maintain separate code bases. In addition to many tablets, notebooks, and desktop computers, UWP is also available on Xbox.



Note The UWP defines a core set of features and functionality. The UWP divides devices into device families: the desktop device family, the mobile device family, the Xbox device family, and so on. Each device family defines the set of APIs and devices on which those APIs are implemented. Additionally, the UWP device family defines a core set of features and functionality that's available across all device families. The libraries available for each device family include conditional methods that enable an app to test which device family it's currently running on.

The purpose of this chapter is to provide a brief description of the concepts that underpin the UWP and to help you get started using Visual Studio 2022 to build apps that operate in this environment. In this chapter, you'll learn about some of the features and tools included with Visual Studio 2022 for building UWP apps, and you will construct an app that conforms to the Windows 11 look and feel. You'll concentrate on learning how to implement a user interface (UI) that scales and adapts to different device resolutions and form factors, and how to apply styling to give the app a distinctive look and feel. Subsequent chapters will focus on the functionality and other features of the app.

You may recall from Chapter 1, "Welcome to C#," that Windows supports several platforms for building GUI apps, from Windows Forms and WPF to MAUI and WinUI 3.0. WinUI 3.0 can run UWP code based on WinRT but can also use the Win32 API to build native Windows apps. It is anticipated that WinUI 3.0 will be the API of choice for building GUI apps on Windows, although it is not yet (at the time of writing) fully available. However, time spent learning UWP is not wasted, as almost everything you'll learn about UWP can be applied to WinUI 3.0. The following image shows how WinUI 3.0 relates to UWP (and WinRT) and the Win32 API:



Note There's not enough space in this book to provide a comprehensive treatise on building UWP apps. Rather, these final chapters concentrate on the basic principles of building an interactive app that uses the Windows UI. For detailed information on writing UWP apps, visit the "What's a Universal Windows Platform (UWP) app?" page on the Microsoft website at <https://docs.microsoft.com/windows/uwp/get-started/universal-application-platform-guide>.

Features of a Universal Windows Platform app

Many modern handheld and tablet devices make it possible for users to interact with apps by using touch. You should design your UWP apps based on this style of user experience (UX).

Windows 11 includes an extensive collection of touch-based controls that also work with a mouse and keyboard. You don't need to separate the touch and mouse features in your apps; simply design your apps for touch, and users can still operate them by using the mouse and keyboard if they prefer or when they're using a device that doesn't support touch interaction.

The way the GUI responds to gestures to provide feedback to the user can greatly enhance the professional feel of your apps. The UWP app templates that come with Visual Studio 2022 include an animation library that you can use in your apps to standardize this feedback and blend in seamlessly with the operating system and software that Microsoft provides.



Note The term *gesture* refers to the manual touch-oriented operations that a user can perform. For example, a user can tap an item with a finger, and the application will typically respond to this gesture the same way that you would expect it to respond to a mouse click or tap. However, gestures can be far more expressive than the simple operations that can be captured by using a mouse. For example, the rotate gesture involves the user placing two fingers on the screen and tracing the arc of a circle with them; in a typical Windows 11 app, this gesture should cause the UI to rotate the selected object in the direction indicated by the movement of the user's fingers. Other gestures include unpinching/pinching to zoom in and out on an item to display more or less detail, pressing and holding to reveal additional information about an item (similar to right-clicking the mouse), and sliding to select an item and drag it across the screen.

The UWP is intended to run on a wide range of devices with varying screen sizes and resolutions. Therefore, when you implement a UWP app, you need to construct your software so that it adapts to the environment it's running in—scaling automatically to the screen size and orientation of the device. This approach opens your software to an increasingly broad market. Additionally, many modern devices can detect their orientation and the speed at which the user changes this orientation through the use of built-in sensors and accelerometers. UWP apps can adapt their layout as the user tilts or rotates a device, making it possible for users to work in a mode that's most comfortable for them.

You should also understand that mobility is a key requirement for many modern apps. With UWP apps, users can roam, and their data can migrate through the cloud to whatever device they happen to be running your app on at a particular moment.

The lifetime of a UWP app is somewhat different from that of a traditional desktop app. You should design apps that can run on devices such as tablets to suspend execution when the user switches focus to another app and then to resume running when the focus returns. This approach can help to conserve resources and battery life on a constrained device. Windows might actually decide to close a suspended app if it determines that it needs to release system resources such as memory. When the app next runs, it should be able to resume where it left off. This means that you need to be prepared to manage app state information in your code, save it to hard disk, and restore it at the appropriate juncture.

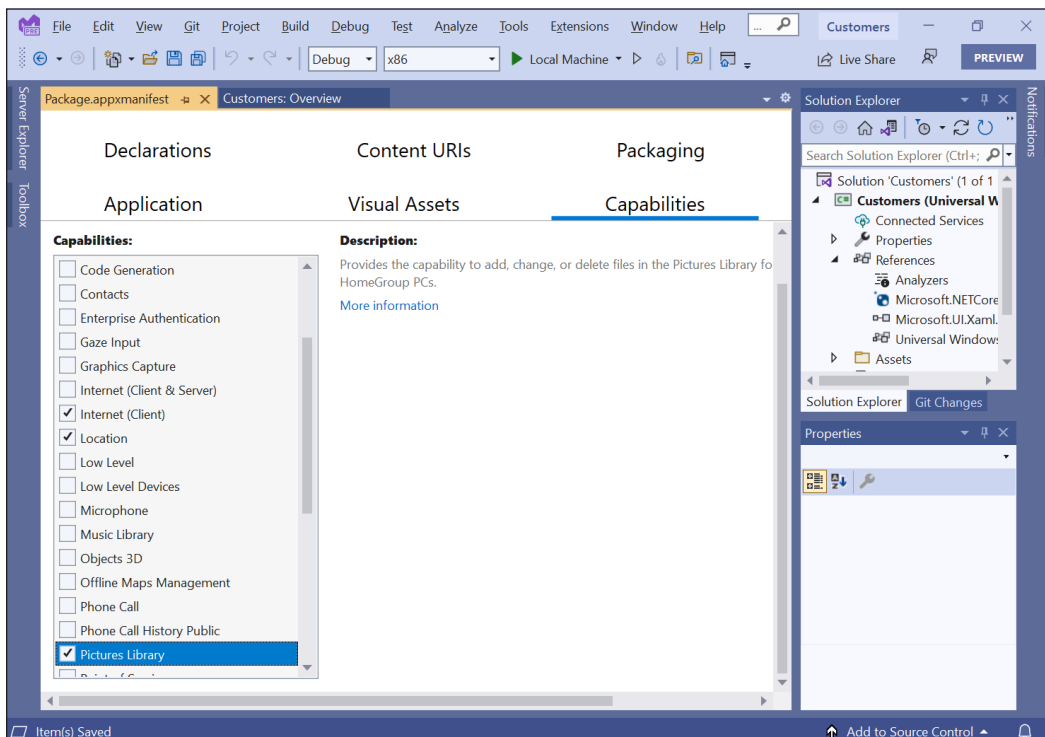


Note You can find more information about how to manage the life cycle of a UWP app on the “Launching, resuming, and background tasks” page on the Microsoft website at <https://docs.microsoft.com/windows/uwp/launch-resume/>.

When you build a new UWP app, you can package it by using the tools provided with Visual Studio 2022 and upload it to the Windows Store. Other users can then connect to the Store, download your app, and install it. You can charge a fee for your app, or you can make it available at no cost. This distribution and deployment mechanism depends on your app being trustworthy and conforming to security policies specified by Microsoft.

When you upload an app to the Windows Store, it undergoes several checks to verify that it does not contain malicious code and that it conforms to the security requirements of a UWP app. These security constraints dictate how your app accesses resources on the computer it’s installed on. For example, by default, a UWP app cannot write directly to the file system or listen for incoming requests from the network—two behaviors commonly exhibited by viruses and other malware.

If your app needs to perform restricted operations, you can specify them as capabilities in the app’s manifest data held in the Package.appxmanifest file. This information is recorded in the metadata of your app and signals Microsoft to perform additional tests to verify the way that your app uses these features. The Package.appxmanifest file is an XML document, but you can edit it in Visual Studio by using the Manifest Designer. The following image shows an example. Here, the Capabilities tab is being used to specify the restricted operations that the application can perform.



In this example, the application declares that it needs to:

- Receive incoming data from the Internet but cannot act as a server and has no local network access.
- Access GPS information that provides information about the location of the device.
- Read and write files held in the user's Pictures folder.

The user is made aware of these requirements and in all cases can disable the settings after installing the app. The application must detect when this has occurred and be prepared to fall back to an alternative solution or disable the functionality that requires these features.



Note You can find more information about the capabilities that UWP apps support on the "App capability declarations" page on the Microsoft website at <https://docs.microsoft.com/windows/uwp/packaging/app-capability-declarations>.

Enough theory; let's get started building a UWP app.

Using the Blank App template to build a Universal Windows Platform app

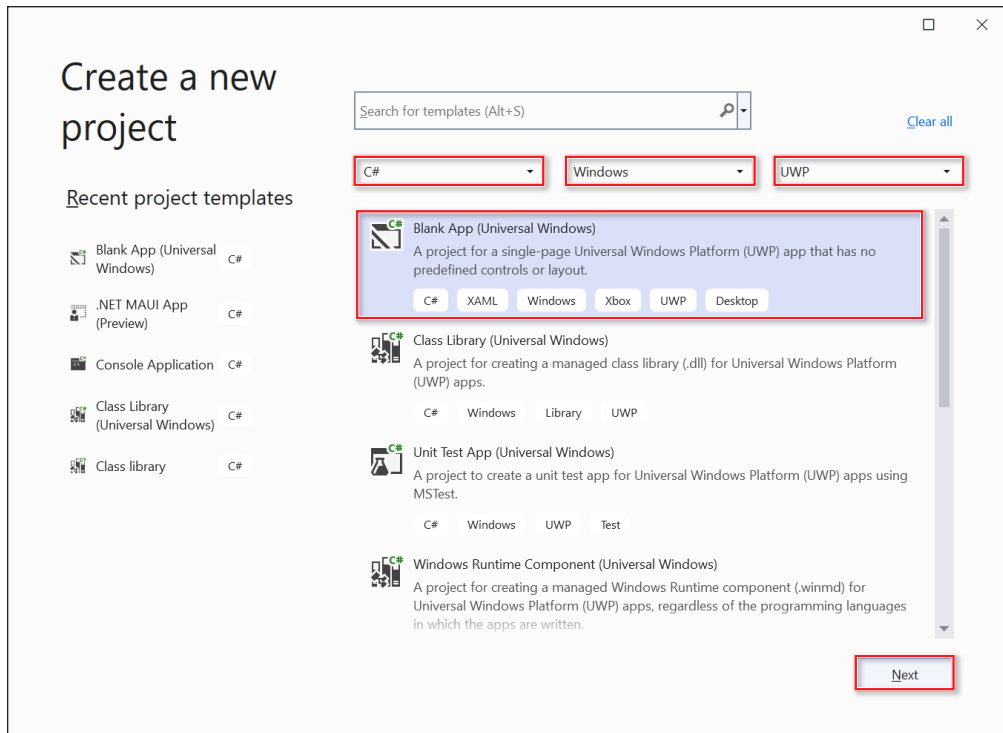
The simplest way to build a UWP app is to use the UWP app templates included with Visual Studio 2022 on Windows 11. Many of the GUI-based applications implemented in earlier chapters have used the Blank App template, and this is a good place to start.

In the following exercises, you'll design the user interface for a simple app for a fictitious company called Adventure Works. This company manufactures and supplies bicycles and associated equipment. The app will enable a user to enter and modify details about the company's customers.

To create the Adventure Works Customers app

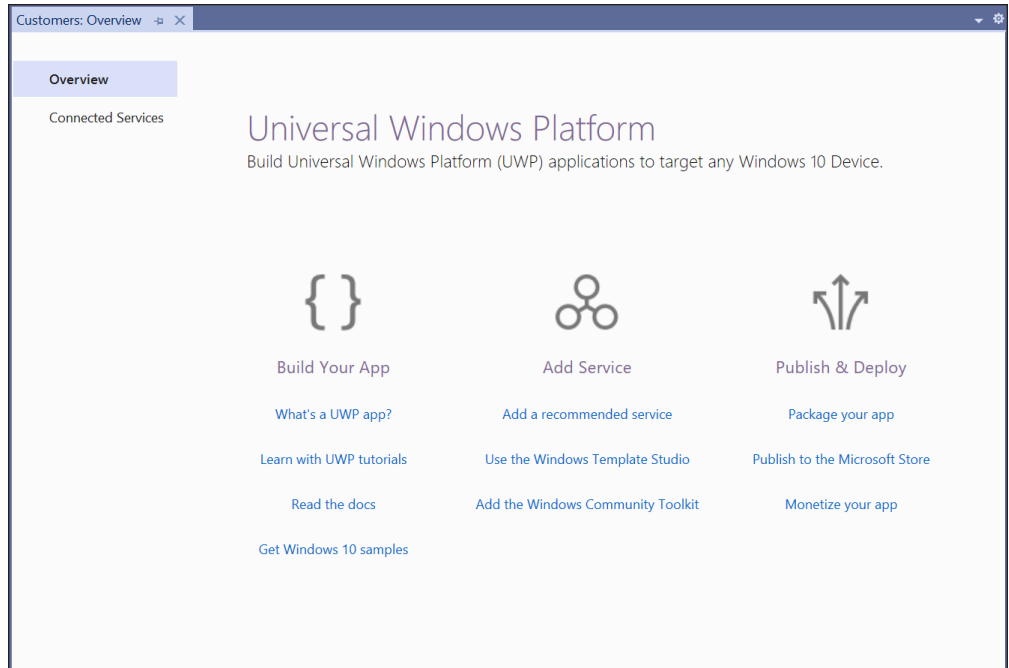
1. If Visual Studio 2022 is not already running, start it, and select **Create a new project**. If Visual Studio 2022 is currently open, on the **File** menu, point to **New**, and then select **Project**.

2. In the Create a new project dialog, set the **Language** dropdown list box to **C#**, the **Platform** dropdown list box to **Windows**, and the **Project Type** dropdown list box to **UWP**. Then select the **Blank App (Universal Windows)** template, and select **Next**.



3. In the Configure your new project dialog, in the **Project name** field, type **Customers**. In the **Location** field, type **\Microsoft Press\VCSBS\Chapter 25** under your **Documents** folder. Then select **Create**.
4. In the New Universal Windows Project dialog, accept the default values in the **Target Version** and **Minimum Version** dropdown list boxes, and then select **OK**.

The new app is created, and the Overview page is displayed. This page contains links to information that you can use to start creating, configuring, and deploying UWP apps.



5. In Solution Explorer, double-click **MainPage.xaml**.

The Design View window appears and displays a blank page. You can drag controls from the Toolbox to add the various controls required by the app, as demonstrated in Chapter 1. However, for this exercise, it's more instructive to concentrate on the XAML markup that defines the layout for the form. If you examine this markup, it should look like this:

```
<Page
    x:Class="Customers.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Customers"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

    <Grid>

    </Grid>
</Page>
```

The form starts with the XAML <Page> tag and finishes with a closing </Page> tag. Everything between these tags defines the content of the page.

The attributes of the <Page> tag contain several declarations of the form `xmlns:id = "..."`. These are XAML namespace declarations, and they operate similarly to C# `using` directives in that they bring items into scope. Many of the controls and other items that you can add to a page are defined in these XAML namespaces, and you can ignore most of these declarations. However, there's one rather curious-looking declaration you should pay attention to:

```
xmlns:local="using:Customers"
```

This declaration brings the items in the C# `Customers` namespace into scope. You can reference classes and other types in this namespace in your XAML code by prefixing them with `local`. The `Customers` namespace is the namespace generated for the code in your app.

6. In Solution Explorer, expand **MainPage.xaml**, and then double-click **MainPage.xaml.cs** to display it in the Code and Text Editor window.

Recall from the exercises earlier in this book that this is the C# file that contains the app logic and event handlers for the form. It looks like this (the `using` directives at the top of the file have been omitted to save space):

```
// The Blank Page item template is documented at http://go.microsoft.com/fwlink/?LinkId=402352&clcid=0x409
namespace Customers
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }
    }
}
```

This file defines the types in the `Customers` namespace. The page is implemented by a class called `MainPage`, and it inherits from the `Page` class. The `Page` class implements the default functionality of an XAML page for a UWP app, so all you have to do is write the code that defines the logic specific to your app in the `MainPage` class.

7. Return to the `MainPage.xaml` file in the Design View window. If you look at the XAML markup for the page, you should notice that the <Page> tag includes the following attribute:

```
x:Class="Customers.MainPage"
```

This attribute connects the XAML markup that defines the layout of the page to the `MainPage` class that provides the logic behind the page.

That's the basic plumbing of a simple UWP app. Of course, what makes a graphical app valuable is the way that it presents information to a user. This isn't always as simple as it sounds. Designing an attractive and easy-to-use GUI requires specialist skills that not all developers have. (I know, because I lack them myself.) However, many graphic artists who do have these skills are not programmers, so although they might be able to design a wonderful user interface, they might not be able to implement the logic required to make it useful. Fortunately, Visual Studio 2022 makes it possible for you to separate the user interface design from the business logic so that a graphic artist and a developer can cooperate to build a really cool-looking app that also works well. All a developer has to do is concentrate on the basic layout of the app and let a graphic artist provide the styling.

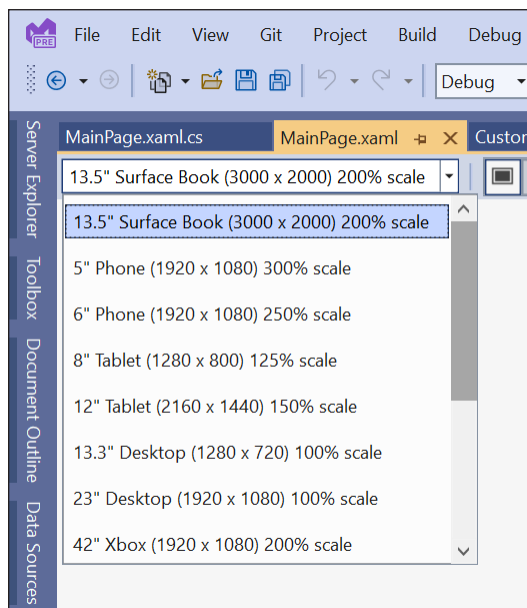
Implementing a scalable user interface

The key to laying out the user interface for a UWP app is to understand how to make it scale and adapt to the different form factors available for the devices users might run the app on. In the following exercises, you'll investigate how to achieve this scaling.

To lay out the page for the Customers app

1. Open the dropdown list box on the left side of the toolbar at the top of the Design View window.

The dropdown list box enables you to select the resolution and form factor of the design surface. There's also a pair of buttons that enable you to select the orientation (portrait or landscape) for devices that support rotations. (Many tablets and other handheld devices do, although desktops, Xbox, Surface Hub, IoT devices, and HoloLens devices don't.) The intent is to enable you to choose different options to quickly see how a user interface will appear on different devices.





Note The default layout is for a Surface Book with a 13.5-inch screen in the landscape orientation. This form factor does not support portrait mode.

2. In the dropdown list box, select **8" Tablet (1280 x 800)**. This is the form factor for a tablet device that supports rotations. Both landscape and portrait modes are available.
3. Select **13.3" Desktop**. This is the form factor that you'll use for the Customers application. This form factor defaults to the landscape orientation.



Note You might find that the page layout in the Design View window appears too small or too large. You can zoom in and out by using the Zoom dropdown list box at the bottom left of the Design View window.

4. Review the XAML markup for the MainPage page. The page contains a single Grid control:

```
<Grid>  
</Grid>
```

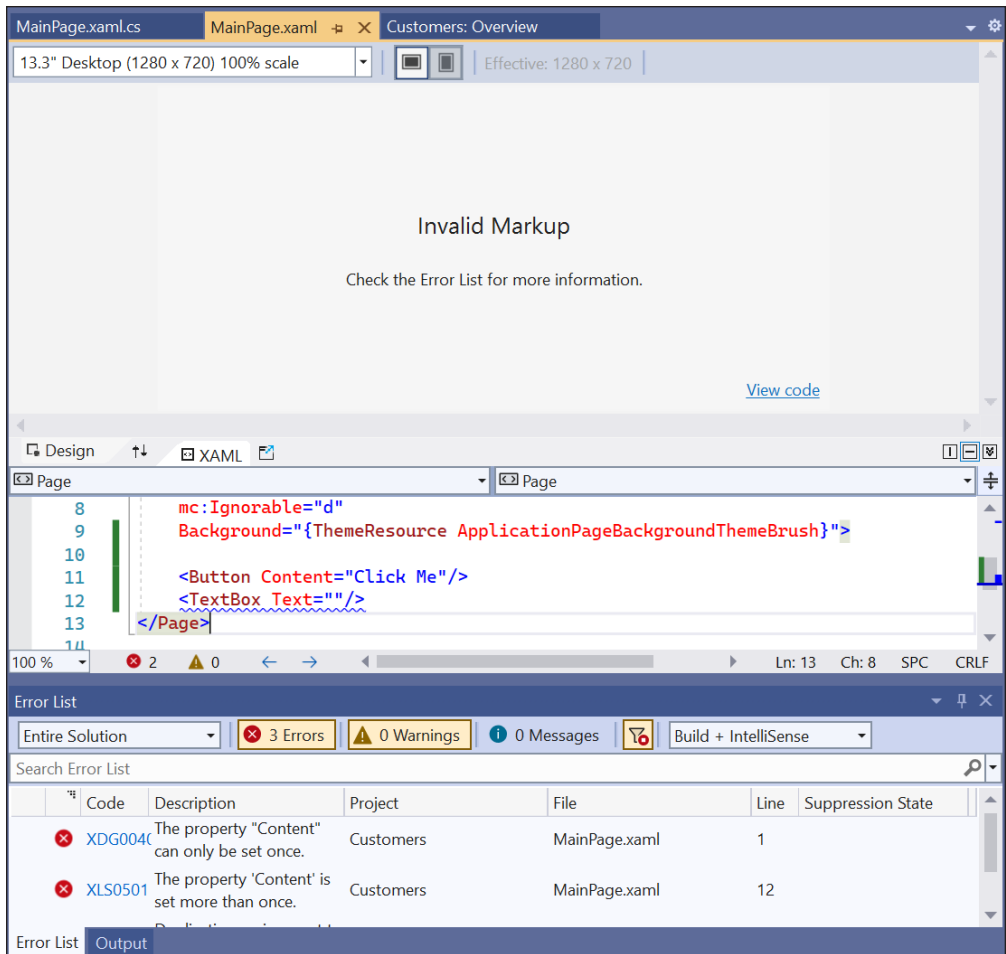
Understanding how the Grid control works is fundamental to building scalable and flexible user interfaces. The Page element can contain only a single item. If you want, you can replace the Grid control with a Button, as shown in the example that follows.



Note Don't type the following code. It is shown for illustrative purposes only.

```
<Page  
    x:Class="Customers.MainPage"  
    ... >  
    <Button Content="Click Me"/>  
</Page>
```

However, the resulting app is probably not very useful; a form that contains a button and that displays nothing else is unlikely to win an award for being the world's greatest app. If you attempt to add a second control, such as a TextBox control, to the page, your code will not compile, and the errors shown in the following image will occur.



The purpose of the `Grid` control is to facilitate adding multiple items to a page. The `Grid` control is an example of a container control; it can contain several other controls, and you can specify the position of these other controls within the grid. Other container controls are also available. For example, the `StackPanel` control automatically places the controls it contains in a vertical arrangement, with each control positioned directly below its immediate predecessor.

In this app, you'll use a `Grid` to hold the controls necessary for a user to be able to enter and view data for a customer.

5. Add a `TextBlock` control to the page, either by dragging it from the **Common XAML Controls** tab in the Toolbox or by typing the text `<TextBlock />` directly into the XAML pane on the blank line after the opening `<Grid>` tag, like this:

```
<Grid>
    <TextBlock />
</Grid>
```




Tip If the Toolbox is not displayed, select Toolbox on the View menu, and it should appear in the toolbar to the left. Select Common XAML Controls to display the contents of the Toolbox. Also, note that you can type the code for a control directly into the XAML window for a page; you don't have to drag controls from the Toolbox.

This TextBlock will provide the title for the page.

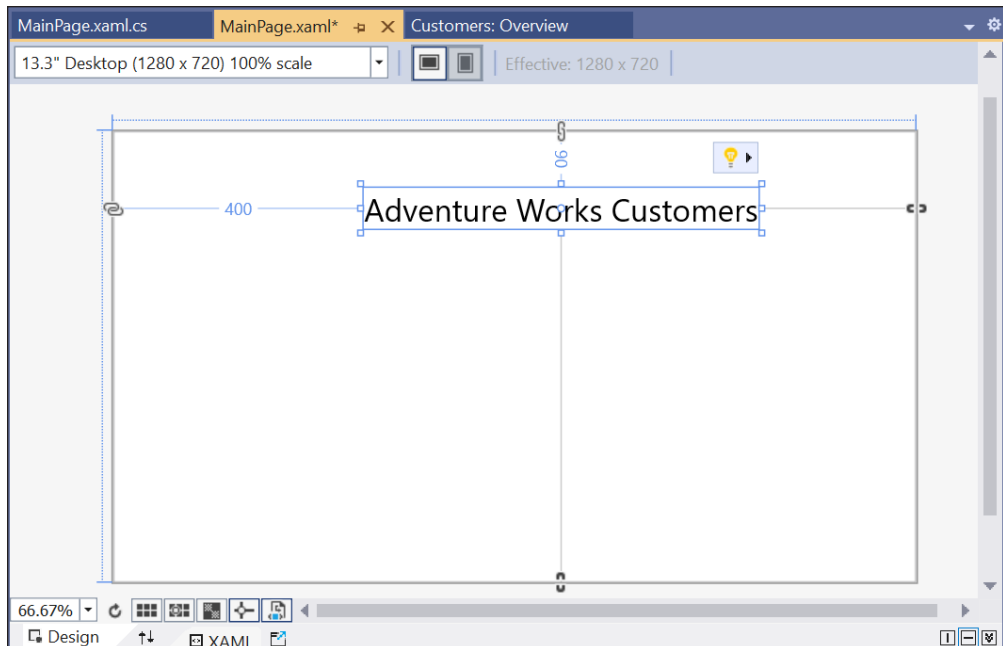
6. Set the properties of the TextBlock control by using the values in the following table:

Property	Value
HorizontalAlignment	Left
Margin	400,90,0,0
TextWrapping	Wrap
Text	Adventure Works Customers
VerticalAlignment	Top
FontSize	50

You can set these properties by using the Properties window or by typing the equivalent XAML markup into the XAML window, as shown here in bold:

```
<TextBlock HorizontalAlignment="Left" Margin="400,90,0,0" TextWrapping="Wrap"  
Text="Adventure Works Customers" VerticalAlignment="Top" FontSize="50"/>
```

The resulting text should appear in the Design View window.



When you drag a control from the Toolbox to a form, connectors appear that specify the distance of two of the sides of the control from the edge of the container control it's placed in. In the preceding example, the connectors for the `TextBlock` control are labeled with the values 400 (from the left edge of the grid) and 90 (from the top edge of the grid). At runtime, if the `Grid` control is resized, the `TextBlock` will move to retain these distances, which in this case might cause the distance of the `TextBlock` in pixels from the right and bottom edges of the `Grid` to change.

You can specify the edge or edges to which a control is anchored by setting the `HorizontalAlignment` and `VerticalAlignment` properties. The `Margin` property specifies the distance from the anchored edges. Again, in this example, the `HorizontalAlignment` property of the `TextBlock` is set to `Left` and the `VerticalAlignment` property is set to `Top`, which is why the control is anchored to the left and top edges of the grid. The `Margin` property contains four values that specify the distance of the left, top, right, and bottom sides (in that order) of the control from the corresponding edge of the container. If one side of a control is not anchored to an edge of the container, you can set the corresponding value in the `Margin` property to 0.

- 7. Add four more `TextBlock` controls to the page. These will be used as labels to help the user identify the data displayed on the page. Use the values in the following table to set the properties of these controls:

Control	Property	Value
First Label	<code>HorizontalAlignment</code>	<code>Left</code>
	<code>Margin</code>	<code>330,190,0,0</code>
	<code>TextWrapping</code>	<code>Wrap</code>
	<code>Text</code>	<code>ID</code>
	<code>VerticalAlignment</code>	<code>Top</code>
	<code>FontSize</code>	<code>20</code>
Second Label	<code>HorizontalAlignment</code>	<code>Left</code>
	<code>Margin</code>	<code>460,190,0,0</code>
	<code>TextWrapping</code>	<code>Wrap</code>
	<code>Text</code>	<code>Title</code>
	<code>VerticalAlignment</code>	<code>Top</code>
	<code>FontSize</code>	<code>20</code>
Third Label	<code>HorizontalAlignment</code>	<code>Left</code>
	<code>Margin</code>	<code>620,190,0,0</code>
	<code>TextWrapping</code>	<code>Wrap</code>
	<code>Text</code>	<code>First Name</code>
	<code>VerticalAlignment</code>	<code>Top</code>
	<code>FontSize</code>	<code>20</code>

Control	Property	Value
Fourth Label	HorizontalAlignment	Left
	Margin	975,190,0,0
	TextWrapping	Wrap
	Text	Last Name
	VerticalAlignment	Top
	FontSize	20

As before, either you can drag the controls from the Toolbox and use the Properties window to set their properties, or you can type the following XAML markup into the XAML pane, after the existing TextBlock control and before the closing </Grid> tag:

```
<TextBlock HorizontalAlignment="Left" Margin="330,190,0,0" TextWrapping="Wrap"
Text="ID" VerticalAlignment="Top" FontSize="20"/>
<TextBlock HorizontalAlignment="Left" Margin="460,190,0,0" TextWrapping="Wrap"
Text="Title" VerticalAlignment="Top" FontSize="20"/>
<TextBlock HorizontalAlignment="Left" Margin="620,190,0,0" TextWrapping="Wrap"
Text="First Name" VerticalAlignment="Top" FontSize="20"/>
<TextBlock HorizontalAlignment="Left" Margin="975,190,0,0" TextWrapping="Wrap"
Text="Last Name" VerticalAlignment="Top" FontSize="20"/>
```

- Below the TextBlock controls, add three TextBox controls to display the text ID, First Name, and Last Name. Use the following table to set the values of these controls. Notice that the Text property should be set to the empty string (""). Also notice that the id TextBox control is marked as read-only. This is because customer IDs will be generated automatically in the code that you add later:

Control	Property	Value
First TextBox	x:Name	id
	HorizontalAlignment	Left
	Margin	300,240,0,0
	TextWrapping	Wrap
	Text	
	VerticalAlignment	Top
	FontSize	20
	IsReadOnly	True
Second TextBox	x:Name	firstName
	HorizontalAlignment	Left
	Margin	550,240,0,0
	TextWrapping	Wrap
	Text	
	VerticalAlignment	Top
	FontSize	20

Control	Property	Value
Third TextBox	x:Name	lastName
	HorizontalAlignment	Left
	Margin	875,240,0,0
	TextWrapping	Wrap
	Text	
	VerticalAlignment	Top
	FontSize	20

The following code shows the equivalent XAML markup for these controls:

```
<TextBox x:Name="id" HorizontalAlignment="Left" Margin="300,240,0,0" TextWrapping="Wrap"
Text="" VerticalAlignment="Top" FontSize="20" IsReadOnly="True"/>
<TextBox x:Name="firstName" HorizontalAlignment="Left" Margin="550,240,0,0"
TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="300" FontSize="20"/>
<TextBox x:Name="lastName" HorizontalAlignment="Left" Margin="875,240,0,0"
TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="300" FontSize="20"/>
```

The Name property is not required for a control but is useful if you want to refer to the control in the C# code for the app. Notice that the Name property is prefixed with x:. This is a reference to the XML namespace <http://schemas.microsoft.com/winfx/2006/xaml> specified in the Page attributes at the top of the XAML markup. This namespace defines the Name property for all controls.



Note It's not necessary to understand why the Name property is defined this way, but for more information, you can read the article "x:Name Directive" at <https://docs.microsoft.com/dotnet/desktop/xaml-services/xname-directive>.

The Width property specifies the width of the control, and the TextWrapping property indicates what happens if the user attempts to enter information into the control that exceeds its width. In this case, all the TextBox controls will wrap the text onto another line of the same width (the control will expand vertically). The alternative value, NoWrap, causes the text to scroll horizontally as the user enters it.

9. Add a ComboBox control to the form, placing it below the Title TextBlock control, between the id and firstName TextBox controls. Set the properties of this control as follows:

Property	Value
x:Name	title
HorizontalAlignment	Left
Margin	420,240,0,0
VerticalAlignment	Top
Width	100
FontSize	20

The equivalent XAML markup for this control is as follows:

```
<ComboBox x:Name="title" HorizontalAlignment="Left" Margin="420,240,0,0"
VerticalAlignment="Top" Width="100" FontSize="20"/>
```

You use a `ComboBox` control to display a list of values from which the user can select.

10. In the XAML pane, replace the definition of the `ComboBox` control and add four `ComboBoxItem` controls, as follows in bold:

```
<ComboBox x:Name="title" HorizontalAlignment="Left" Margin="420,240,0,0"
  VerticalAlignment="Top" Width="75" FontSize="20">
  <ComboBoxItem Content="Mr"/>
  <ComboBoxItem Content="Mrs"/>
  <ComboBoxItem Content="Ms"/>
  <ComboBoxItem Content="Miss"/>
</ComboBox>
```

The `ComboBoxItem` elements are displayed in a dropdown list when the app runs, and the user can select one of them.

There's one important syntactical point to notice in this code: the `ComboBox` markup has been split into an opening `<ComboBox>` tag and a closing `</ComboBox>` tag. You place the `ComboBoxItem` controls between these opening and closing tags.



Note A `ComboBox` control can display simple elements such as a set of `ComboBoxItem` controls that display text, but it can also contain more complex elements such as buttons, checkboxes, and option buttons. If you're adding simple `ComboBoxItem` controls, it's probably easier to type the XAML markup by hand, but if you're adding more complex controls, the Object Collection Editor available in the Properties window can prove very useful. However, you should avoid trying to be too clever in a combo box; the best apps are those that provide the most intuitive UIs, and embedding complex controls in a combo box can be confusing to a user.

11. Add two more `TextBox` controls and two more `TextBlock` controls to the form. In the `TextBox` controls, the user will be able to enter an email address and a telephone number for the customer. The `TextBlock` controls provide the labels for the text boxes. Use the values in the following table to set the properties of the controls:

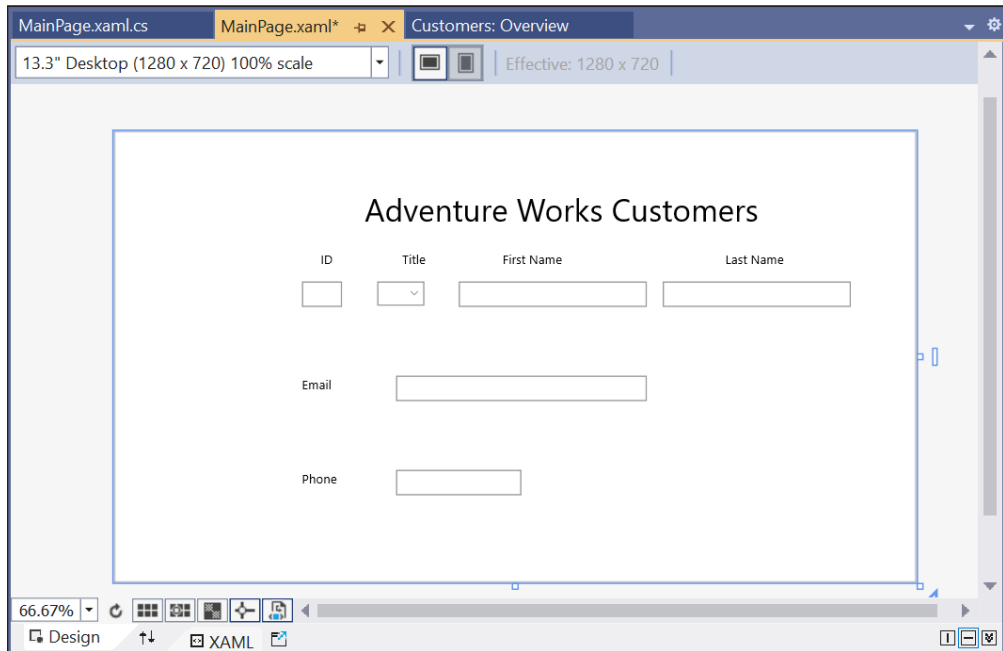
Control	Property	Value
First TextBlock	HorizontalAlignment	Left
	Margin	300, 390, 0, 0
	TextWrapping	Wrap
	Text	Email
	VerticalAlignment	Top
	FontSize	20

Control	Property	Value
First TextBox	x:Name	email
	HorizontalAlignment	Left
	Margin	450,390,0,0
	TextWrapping	Wrap
	Text	Leave Empty
	VerticalAlignment	Top
	Width	400
	FontSize	20
Second TextBlock	HorizontalAlignment	Left
	Margin	300,540,0,0
	TextWrapping	Wrap
	Text	Phone
	VerticalAlignment	Top
	FontSize	20
Second TextBox	x:Name	phone
	HorizontalAlignment	Left
	Margin	450,540,0,0
	TextWrapping	Wrap
	Text	Leave Empty
	VerticalAlignment	Top
	Width	200
	FontSize	20

The XAML markup for these controls should look like this:

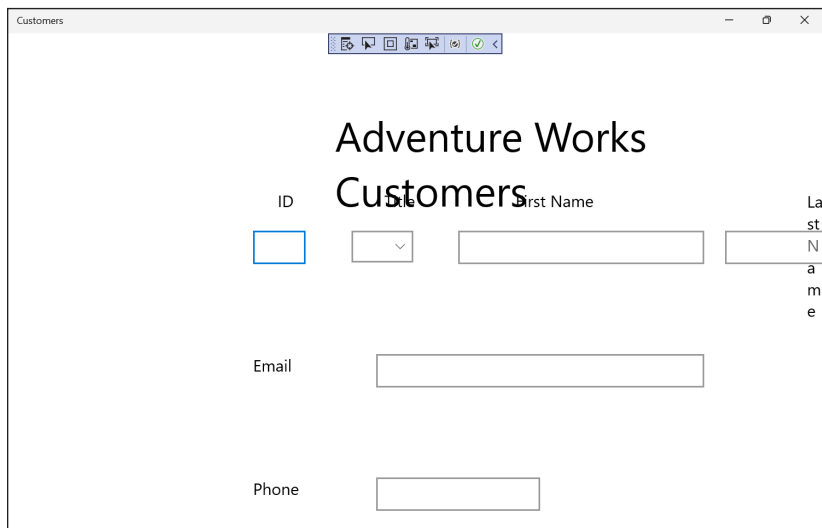
```
<TextBlock HorizontalAlignment="Left" Margin="300,390,0,0" TextWrapping="Wrap"
Text="Email" VerticalAlignment="Top" FontSize="20"/>
<TextBox x:Name="email" HorizontalAlignment="Left" Margin="450,390,0,0"
TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="400" FontSize="20"/>
<TextBlock HorizontalAlignment="Left" Margin="300,540,0,0" TextWrapping="Wrap"
Text="Phone" VerticalAlignment="Top" FontSize="20"/>
<TextBox x:Name="phone" HorizontalAlignment="Left" Margin="450,540,0,0"
TextWrapping="Wrap" Text="" VerticalAlignment="Top" Width="200" FontSize="20"/>
```

The completed form in the Design View window should look like the following image.



12. On the **Debug** menu, select **Start Debugging** to build and run the app.

The app starts and displays the form. You can enter data into the form and select a title from the combo box, but you can't do much else yet. However, a much bigger problem is that, depending on the resolution of your screen, the form looks awful. (If the form looks fine, drag the right-hand edge to make it narrower.) The right side of the display has been cut off, much of the text has wrapped around, and the Last Name text box has been truncated.



13. Select and drag the right side of the window to expand the display so that the text and controls are displayed as they appeared in the Design View window in Visual Studio. This is the optimal size of the form as it was designed.
14. Resize the window displaying the Customers app to its minimum width. This time, much of the form disappears. Some of the TextBlock content wraps, but the form is clearly not usable in this view.
15. Return to Visual Studio, and on the **Debug** menu, select **Stop Debugging**.

That was a salutary lesson in being careful about how you lay out an app. Although the app looked fine when it ran in a window that was the same size as the Design View, as soon as you resized the window to a narrower view, it became less useful (or even completely useless). Additionally, the app assumes that the user will be viewing the screen on a device in the landscape orientation. If you temporarily switch the Design View window to the 12" Tablet form factor and select the Portrait orientation button, you can see what the form would look like if the user ran the app on a tablet that supports different orientations and rotated the device to switch to portrait mode. (Don't forget to switch back to the 13.3" Desktop form factor afterward.)

The issue is that the layout technique shown so far does not scale and adapt to different form factors and orientations. Fortunately, you can use the properties of the Grid control and another feature called the Visual State Manager to solve these problems.

Implementing a tabular layout by using a Grid control

You can use the Grid control to implement a tabular layout. A Grid control contains rows and columns, and you can specify which rows and columns other controls should be placed in.

The beauty of the Grid control is that you can specify the sizes of the rows and columns that it contains as relative values. That way, if the grid shrinks or grows to adapt itself to the different form factors and orientations that users might switch to, the rows and columns can shrink and grow in proportion to the grid. The intersection of a row and a column in a grid defines a cell, and if you position controls in cells, they will move as the rows and columns shrink and grow. Therefore, the key to implementing a scalable UI is to break it down into a collection of cells and place related elements in the same cell. A cell can even contain another grid, giving you the ability to fine-tune the exact positioning of each element.

If you consider the Customers app, you can see that the UI breaks down into two main areas: a heading containing the title and the body containing the customers' details. Allowing for some spacing between these areas and a margin at the bottom of the form, you can assign relative sizes to each of these areas, as shown in the following diagram.

Adventure Works Customers				2x
				x
ID	Title	First Name	Last Name	10x
Email				
Phone				2x

The diagram shows only rough approximations, but the row for the heading is twice as high as the row for the spacer below it. The row for the body is 10 times as high as the spacer, and the bottom margin is twice the height of the spacer.

To hold the elements in each area, you can define a grid with four rows and place the appropriate items in each row. However, the body of the form can be described by another, more complex grid, as shown in the following diagram:

	ID		Title		First Name		Last Name		y
									y
									2y
	Email								y
									2y
	Phone								y
									4y
z	z		z		2z		2z	z	

Again, the height of each row is specified in relative terms, as is the width of each column. Also, you can clearly see that the TextBox elements for Email and Phone do not quite fit this grid pattern.

If you were being pedantic, you might choose to define further grids inside the body of the form to make these items fit. However, you should keep in mind the purpose of this grid: to define the relative positioning and spacing of elements. Therefore, it is acceptable for an element to extend beyond the boundaries of a cell in the grid arrangement.

In the next exercise, you'll modify the layout of the Customers app to use this grid format to position the controls.

To modify the layout to scale to different form factors and orientations

1. In the XAML pane for the Customers app, add another `Grid` inside the existing `Grid` control, before the first `TextBlock` control. Give this new `Grid` control a margin of 10 pixels from the left and right edges of the parent `Grid` control and 20 pixels from the top and bottom, as shown in bold in the following code:

```
<Grid>
    <Grid Margin="10,20,10,20">
    </Grid>
    <TextBlock HorizontalAlignment="Left" TextWrapping="Wrap"
Text="Adventure Works Customers" ... />
    ...
</Grid>
```



Note You could define the rows and columns as part of the existing `Grid` control, but to maintain a consistent look and feel with other UWP apps, you should leave some blank space to the left and at the top of a page.

2. Add the following `<Grid.RowDefinitions>` section shown in bold to the new `Grid` control:

```
<Grid Margin="10,20,10,20">
    <Grid.RowDefinitions>
        <RowDefinition Height="2*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="10*" />
        <RowDefinition Height="2*" />
    </Grid.RowDefinitions>
</Grid>
```

The `<Grid.RowDefinitions>` section defines the rows for the grid. In this example, you have defined four rows. You can specify the size of a row as an absolute value in pixels, or you can use the `*` operator to indicate that the sizes are relative and that Windows should calculate the row sizes itself when the app runs, depending on the form factor and resolution of the screen. The values used in this example correspond to the relative row sizes for the header, body, spacer, and bottom margin of the Customers form shown in the earlier diagram.

3. Move the `TextBlock` control that contains the text `Adventure Works Customers` into the `Grid` control directly after the closing `</Grid.RowDefinitions>` tag but before the closing `</Grid>` tag.

4. Add a `Grid.Row` attribute to the `TextBlock` control and set the value to 0.

```
<Grid Margin="10,20,10,20">
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" ... Text="Adventure Works Customers" ... />
    ...
</Grid>
```

This indicates that the `TextBlock` element should be positioned within the first row of the `Grid` control. (Grid controls number rows and columns starting at 0.)



Note The `Grid.Row` attribute is an example of an attached property. An *attached* property is one that a control receives from the container control that it's placed in. Outside a grid, a `TextBlock` control doesn't have a `Row` property, as it would be meaningless. But when positioned within a grid, the `Row` property is attached to the `TextBlock` control, and the `TextBlock` control can assign it a value. The `Grid` control then uses this value to determine where to display the `TextBlock` control. Attached properties are easy to spot because they have the form `ContainerType.PropertyName`.

5. Remove the `Margin` property of the `TextBlock` control and set the `HorizontalAlignment` and `VerticalAlignment` properties to `Center`. This will cause the `TextBlock` to appear centered in the row. The XAML markup for the `Grid` and `TextBlock` controls should look like this (the changes to the `TextBlock` are highlighted in bold):

```
<Grid Margin="10,20,10,20">
    ...
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" HorizontalAlignment="Center" TextWrapping="Wrap"
Text="Adventure Works Customers" VerticalAlignment="Center" FontSize="50"/>
    ...
</Grid>
```

6. After the `TextBlock` control, add another nested `Grid` control. This `Grid` control will be used to lay out the controls in the body of the form. It should appear in the third row of the outer `Grid` control (the row of size 10*), so set the `Grid.Row` property to 2, as shown in bold in the following code:

```
<Grid Margin="10,20,10,20">
    <Grid.RowDefinitions>
        <RowDefinition Height="2*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="10*" />
        <RowDefinition Height="2*" />
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" HorizontalAlignment="Center" ... />
    <Grid Grid.Row="2">
        ...
    </Grid>
    ...
</Grid>
```

7. Add the following `<Grid.RowDefinitions>` and `<Grid.ColumnDefinitions>` sections to the new Grid control:

```
<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="2*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="2*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="4*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="20" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="20" />
        <ColumnDefinition Width="2*" />
        <ColumnDefinition Width="20" />
        <ColumnDefinition Width="2*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
</Grid>
```

These row and column definitions specify the height and width of each of the rows and columns shown earlier in the diagram that depicted the structure of the body of the form. There is a small space of 20 pixels between each of the columns that will hold controls.

8. Move the `TextBlock` controls that display the ID, Title, First Name, and Last Name labels inside the nested Grid control, immediately after the closing `<Grid.ColumnDefinitions>` tag.
9. Set the `Grid.Row` property for each `TextBlock` control to 0 (these labels will appear in the first row of the grid). Then set the `Grid.Column` property for the ID label to 1, the `Grid.Column` property for the Title label to 3, the `Grid.Column` property for the First Name label to 5, and the `Grid.Column` property for the Last Name label to 7.
10. Remove the `Margin` property from each of the `TextBlock` controls and set the `HorizontalAlignment` and `VerticalAlignment` properties to `Center`. The XAML markup for these controls should look like this (the changes are highlighted in bold):

```
<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        ...
    </Grid.ColumnDefinitions>
    <TextBlock Grid.Row="0" Grid.Column="1" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="ID" VerticalAlignment="Center" FontSize="20"/>
    <TextBlock Grid.Row="0" Grid.Column="3" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Title" VerticalAlignment="Center" FontSize="20"/>
    <TextBlock Grid.Row="0" Grid.Column="5" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="First Name" VerticalAlignment="Center" FontSize="20"/>
```

```
<TextBlock Grid.Row="0" Grid.Column="7" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Last Name" VerticalAlignment="Center" FontSize="20"/>
</Grid>
```

11. Move the `id`, `firstName`, and `lastName` `TextBox` controls and the `title` `ComboBox` control inside the nested `Grid` control, immediately after the `Last Name` `TextBlock` control. Place these controls in row 1 of the `Grid` control. Put the `id` control in column 1, the `title` control in column 3, the `firstName` control in column 5, and the `lastName` control in column 7.
12. Remove the `Margin` of each of these controls and set the `VerticalAlignment` property to `Center`. Remove the `Width` property and set the `HorizontalAlignment` property to `Stretch`. This causes the control to occupy the entire cell when it is displayed, and the control shrinks or grows as the size of the cell changes.

The completed XAML markup for these controls should look like this, with changes highlighted in bold:

```
<Grid Grid.Row="2">
<Grid.RowDefinitions>
...
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
...
</Grid.ColumnDefinitions>
...
<TextBlock Grid.Row="0" Grid.Column="7" ... Text="Last Name" .../>
<TextBox Grid.Row="1" Grid.Column="1" x:Name="id" HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20" IsReadOnly="True"/>
<TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName" HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20"/>
<TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName" HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20"/>
<ComboBox Grid.Row="1" Grid.Column="3" x:Name="title" HorizontalAlignment="Stretch"
VerticalAlignment="Center" FontSize="20">
<ComboBoxItem Content="Mr"/>
<ComboBoxItem Content="Mrs"/>
<ComboBoxItem Content="Ms"/>
<ComboBoxItem Content="Miss"/>
</ComboBox>
</Grid>
```

13. Move the `TextBlock` control for the `Email` label and the `email` `TextBox` control to the nested `Grid` control, immediately after the closing tag of the `title` `ComboBox` control. Place these controls in row 3 of the `Grid` control. Put the `Email` label in column 1 and the `email` `TextBox` control in column 3. Additionally, set the `Grid.ColumnSpan` property for the `email` `TextBox` control to 5; this way, the column can spread to the value specified by its `Width` property across five columns, as shown in the earlier diagram.
14. Set the `HorizontalAlignment` property of the `Email` label control to `Center`, but leave the `HorizontalAlignment` property of the `email` `TextBox` set to `Left`. This control should remain left-justified against the first column that it spans rather than being centered across them all.

15. Set the `VerticalAlignment` property of the Email label and the email `TextBox` control to `Center`. Then remove the `Margin` property for both of these controls.

The following XAML markup shows the completed definitions of these controls:

```
<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        ...
    </Grid.ColumnDefinitions>
    ...
    <ComboBox Grid.Row="1" Grid.Column="3" x:Name="title" ...>
        ...
    </ComboBox>
    <TextBlock Grid.Row="3" Grid.Column="1" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Email" VerticalAlignment="Center" FontSize="20"/>
    <TextBox Grid.Row="3" Grid.Column="3" Grid.ColumnSpan="5" x:Name="email"
HorizontalAlignment="Left" TextWrapping="Wrap" Text="" VerticalAlignment="Center"
Width="400" FontSize="20"/>
</Grid>
```

16. Move the `TextBlock` control for the Phone label and phone `TextBox` control to the nested `Grid` control, immediately after the email `TextBox` control. Place these controls in row 5 of the `Grid` control. Put the Phone label in column 1 and the phone `TextBox` control in column 3. Set the `Grid.ColumnSpan` property for the phone `TextBox` control to 3.
17. Set the `HorizontalAlignment` property of the Phone label control to `Center`, and leave the `HorizontalAlignment` property of the phone `TextBox` set to `Left`. Then set the `VerticalAlignment` property of both controls to `Center`, and remove the `Margin` property.

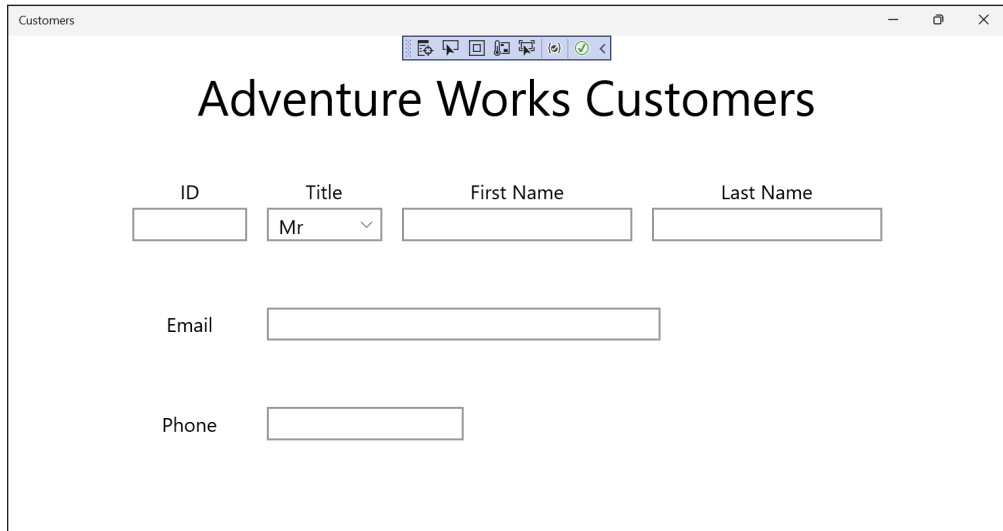
The following XAML markup shows the completed definitions of these controls:

```
<Grid Grid.Row="2">
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        ...
    </Grid.ColumnDefinitions>
    ...
    <TextBox ... x:Name="email" .../>
    <TextBlock Grid.Row="5" Grid.Column="1" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Phone" VerticalAlignment="Center" FontSize="20"/>
    <TextBox Grid.Row="5" Grid.Column="3" Grid.ColumnSpan="3" x:Name="phone"
HorizontalAlignment="Left" TextWrapping="Wrap" Text="" VerticalAlignment="Center"
Width="200" FontSize="20"/>
</Grid>
```

In the next exercise, you'll run the app to see how the layout adapts to different resolutions and form factors.

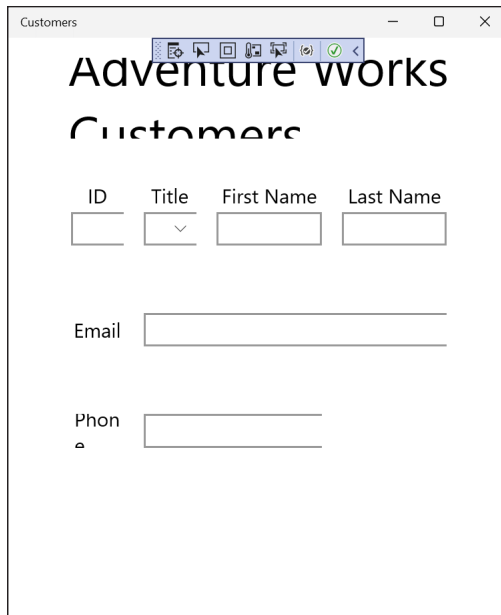
To test the app

1. On the **Debug** menu, select **Start Debugging**. The app should look similar to the image that follows:



A screenshot of a Windows application window titled "Customers". The window has a standard Windows title bar with minimize, maximize, and close buttons. Below the title bar is a toolbar with several icons. The main content area displays the title "Adventure Works Customers" in a large, bold, black font. Below the title, there are four input fields arranged horizontally: "ID", "Title" (with a dropdown menu showing "Mr"), "First Name", and "Last Name". Below these fields, there are two more input fields: "Email" and "Phone", each on a new line.

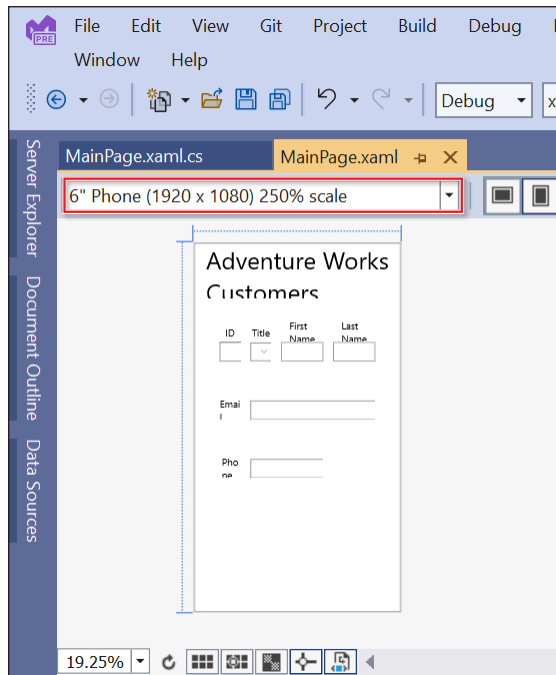
2. Resize the window to make it narrower. The controls should adjust their positions and sizes. However, at some point, the window will become too narrow to display the text and the fixed-width controls (email and phone) correctly, causing text wrapping and truncation. This is how the app might appear on a small tablet or phone:



A screenshot of the same "Customers" application window, but resized to be much narrower. The title "Adventure Works Customers" is now wrapped onto two lines. The input fields are also affected: the "ID", "Title", "First Name", and "Last Name" fields are still visible but their spacing is compressed. The "Email" and "Phone" fields are truncated, with only "Email" and "Phon" visible respectively. The overall layout is cramped and text is cut off.

3. Return to Visual Studio and stop debugging.
4. In the dropdown list box at the top of the Design View window, select the **6" Phone** layout.

The Design View window adjusts the display and confirms how the app might appear on a phone device.



5. Experiment with the other device sizes and orientations. Notice how the grid causes the controls to adapt their positions.
6. Switch back to the **13.3" Desktop** layout before continuing.

Adapting the layout by using the Visual State Manager

The user interface for the Customers app scales for different resolutions and form factors, but it still doesn't work well if you reduce the width of the view, and it probably wouldn't look too good on a phone or other narrow device. If you think about it, the solution to the problem in these cases is not so much a matter of scaling the controls as actually laying them out in a different way. For example, it would make better sense if the Customers form looked like the following image in a narrow view.

Customers

Customers

ID

Title

First Name

Last Name

Email

Phone

You can achieve this effect in several ways:

- **By creating several versions of the MainPage.xaml file, one for each device family**
Each of these XAML files can be linked to the same underlying code (MainPage.xaml.cs) so that they all run the same code. For example, to create an XAML file for a smartphone, add a folder named DeviceFamily-Mobile (this name is important) to the project and then use the Add New Item menu command to add a new XAML view named MainPage.xaml to the folder. Lay out the controls on this page folder as they should be displayed on a smartphone. The XAML view will be linked automatically to the existing MainPage.xaml.cs file. At runtime, the UWP will select the appropriate view based on the type of device that the app is running on.
- **By using the Visual State Manager to modify the layout of the page at runtime** All UWP apps implement a Visual State Manager that tracks the visual state of an app. It can detect when the height and width of the window changes, and you can add XAML markup that positions controls depending on the size of the window. This markup can move controls around or display and hide controls.
- **By using the Visual State Manager to switch between views based on the height and width of the window** This approach is a hybrid of the first two options described here, but it's the least messy (you don't have to write lots of tricky code to calculate the best position for each control) and the most flexible (it will work if the window is narrowed on the same device).

You'll use the third approach in the next exercises. The first step is to define a layout for the customers' data that should appear in a narrow view.

To define a layout for the narrow view

1. In the XAML pane for the Customers app, add the `x:Name` and `Visibility` properties shown here in bold to the first nested `Grid` control:

```
<Grid">
    <Grid x:Name="customersTabularView" Margin="10,20,10,20" Visibility="Collapsed">
        ...
    </Grid>
</Grid>
```

This `Grid` control will hold the default view of the form. You'll reference this `Grid` control in other XAML markup later in this set of exercises—thus the requirement to give it a name. The `Visibility` property specifies whether the control is displayed (`Visible`) or hidden (`Collapsed`). The default value is `Visible`, but for the time being, you'll hide this `Grid` control while you define another one to display the data in a columnar format.

2. After the closing `</Grid>` tag for the `customersTabularView` `Grid` control, add another `Grid` control. Set the `x:Name` property to `customersColumnarView`, set the `Margin` property to `10,20,10,20`, and set the `Visibility` property to `Visible`.



Tip You can expand and contract elements in the XAML pane of the Design View window and make the structure easier to read by selecting the + and – signs that appear down the left edge of the XAML markup.

```
<Grid>
    <Grid x:Name="customersTabularView" Margin="10,20,10,20" Visibility="Collapsed">
        ...
    </Grid>
    <Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Visible">
        </Grid>
</Grid>
```

This `Grid` control will hold the “narrow” view of the form. The fields in this grid will be laid out in a columnar manner as described earlier.

3. In the `customersColumnarView` `Grid` control, add the following row definitions:

```
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Visible">
    <Grid.RowDefinitions>
        <RowDefinition Height="*/"/>
        <RowDefinition Height="10*/"/>
    </Grid.RowDefinitions>
</Grid>
```

You'll use the top row to display the title and the second, much larger row to display the controls in which users enter data.

4. Immediately after the row definitions, add the **TextBlock** control shown here in bold. This control displays a truncated title, **Customers**, in the first row of the **Grid** control. Set **FontSize** to 30.

```
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Visible">
    <Grid.RowDefinitions>
        ...
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" HorizontalAlignment="Center" TextWrapping="Wrap"
Text="Customers" VerticalAlignment="Center" FontSize="30"/>
</Grid>
```

5. Add another **Grid** control to row 1 of the **customersColumnarView** **Grid** control, directly after the **TextBlock** control that contains the **Customers** title. This **Grid** control will display the labels and data-entry controls in two columns, so add the row and columns definitions shown in bold in the following code example to this **Grid** control:

```
<TextBlock Grid.Row="0" ... />
<Grid Grid.Row="1">
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
</Grid>
```



Note If all the rows or columns in a set have the same height or width, you don't need to specify their size.

6. Copy the XAML markup for the **ID**, **Title**, **First Name**, and **Last Name** **TextBlock** controls from the **customersTabularView** **Grid** control to the new **Grid** control, immediately after the row definitions that you just added. Put the **ID** control in row 0, the **Title** control in row 1, the **First Name** control in row 2, and the **Last Name** control in row 3. Place all controls in column 0.

```
<Grid.RowDefinitions>
    ...
</Grid.RowDefinitions>
<TextBlock Grid.Row="0" Grid.Column="0" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="ID" VerticalAlignment="Center" FontSize="20"/>
<TextBlock Grid.Row="1" Grid.Column="0" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Title" VerticalAlignment="Center" FontSize="20"/>
<TextBlock Grid.Row="2" Grid.Column="0" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="First Name" VerticalAlignment="Center" FontSize="20"/>
<TextBlock Grid.Row="3" Grid.Column="0" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Last Name" VerticalAlignment="Center" FontSize="20"/>
```

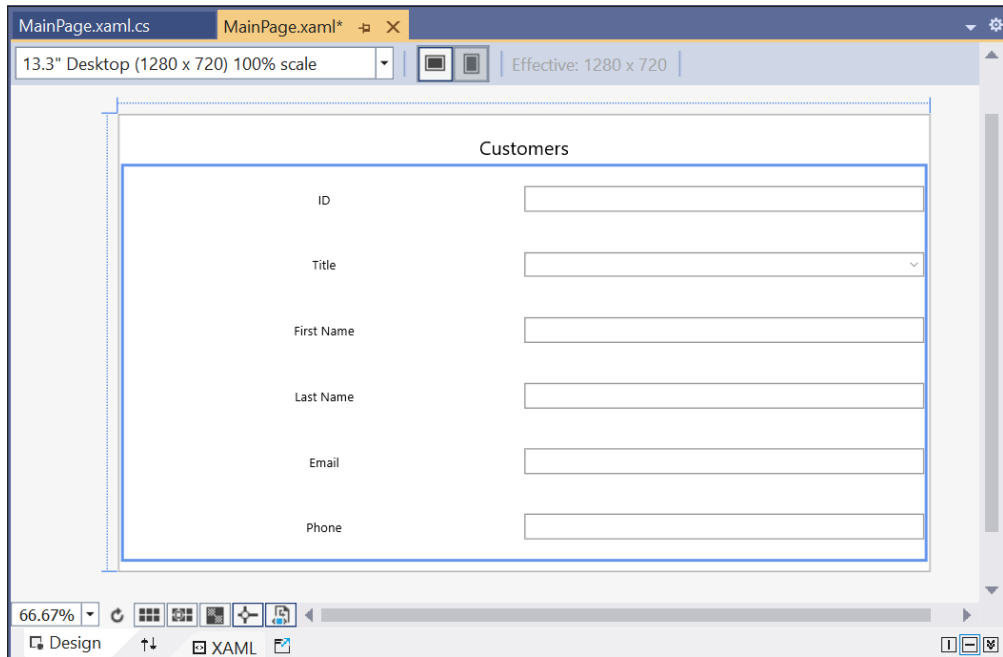
7. Copy the XAML markup for the `id`, `title`, `firstName`, and `lastName` `TextBox` and `ComboBox` controls from the `customersTabularViewGrid` control to the new `Grid` control, immediately after the `TextBox` controls. Put the `id` control in row 0, the `title` control in row 1, the `firstName` control in row 2, and the `lastName` control in row 3. Place all four controls in column 1. Also, change the names of the controls by prefixing them with the letter `c` (for column). This final change is necessary to avoid clashing with the names of the existing controls in the `customersTabularViewGrid` control.

```
<TextBlock Grid.Row="3" Grid.Column="0" HorizontalAlignment="Center"
TextWrapping="Wrap" Text="Last Name" .../>
<TextBox Grid.Row="0" Grid.Column="1" x:Name="cId" HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20" IsReadOnly="True"/>
<TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName" HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20"/>
<TextBox Grid.Row="3" Grid.Column="1" x:Name="cLastName" HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20"/>
<ComboBox Grid.Row="1" Grid.Column="1" x:Name="cTitle" HorizontalAlignment="Stretch"
VerticalAlignment="Center" FontSize="20">
    <ComboBoxItem Content="Mr"/>
    <ComboBoxItem Content="Mrs"/>
    <ComboBoxItem Content="Ms"/>
    <ComboBoxItem Content="Miss"/>
</ComboBox>
```

8. Copy the `TextBlock` and `TextBox` controls for the email address and telephone number from the `customersTabularViewGrid` control to the new `Grid` control, placing them after the `cTitle` `ComboBox` control. Place the `TextBlock` controls in column 0, rows 4 and 5, and the `TextBox` controls in column 1, rows 4 and 5. Change the name of the email `TextBox` control to `cEmail` and the name of the phone `TextBox` control to `cPhone`. Remove the `Width` properties of the `cEmail` and `cPhone` controls, and set their `HorizontalAlignment` properties to `Stretch`.

```
<ComboBox ...>
...
</ComboBox>
<TextBlock Grid.Row="4" Grid.Column="0" HorizontalAlignment="Center" TextWrapping="Wrap"
Text="Email" VerticalAlignment="Center" FontSize="20"/>
<TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail" HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20"/>
<TextBlock Grid.Row="5" Grid.Column="0" HorizontalAlignment="Center" TextWrapping="Wrap"
Text="Phone" VerticalAlignment="Center" FontSize="20"/>
<TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone" HorizontalAlignment="Stretch"
TextWrapping="Wrap" Text="" VerticalAlignment="Center" FontSize="20"/>
```

The Design View window should display the columnar layout like this:



9. Return to the XAML markup for the `customersTabularViewGrid` control and set the `Visibility` property to `Visible` as follows:

```
<Grid x:Name="customersTabularView" Margin="10,20,10,20" Visibility="Visible">
```

10. In the XAML markup for the `customersColumnarViewGrid` control, set the `Visibility` property to `Collapsed` as follows:

```
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Collapsed">
```

The Design View window should display the original tabular layout of the Customers form. This is the default view that will be used by the app.

You have now defined the layout that will appear in the narrow view. You might be concerned that in essence all you have done is duplicated many of the controls and laid them out in a different manner. If you run the form and switch between views, how will data in one view transfer to the other? For example, if you enter the details for a customer when the app is running full screen and then you switch to the narrow view, the newly displayed controls will not contain the same data that you just entered. UWP apps address this problem by using *data binding*. This is a technique by which you can associate the same piece of data to multiple controls. As the data changes, all controls display the updated information. You'll see how this works in Chapter 26, "Displaying and searching for data in a Universal Windows Platform app." For the time being, you need to consider only how to use the Visual State Manager to switch between layouts when the view changes.

You can use triggers that alert the Visual State Manager when some aspect (such as the height or width) of the display changes. You can define the visual state transitions performed by these triggers in the XAML markup of your app. This is what you'll do in the next exercise.

To use the Visual State Manager to modify the layout

1. In the **XAML** pane for the Customers app, after the closing `</Grid>` tag for the customers ColumnarView Grid control, add the following markup shown in bold:

```
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Visible">
    ...
</Grid>
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
        <VisualState x:Name="TabularLayout">
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
```

You define the visual state transitions by implementing one or more visual state groups. Each visual state group specifies the transitions that should occur when the Visual State Manager switches to this state. Each state should be given a meaningful name to help you identify its purpose.

2. Add the following visual state trigger shown in bold to the visual state group:

```
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
        <VisualState x:Name="TabularLayout">
            <VisualState.StateTriggers>
                <AdaptiveTrigger MinWindowWidth="660"/>
            </VisualState.StateTriggers>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

This trigger will fire whenever the width of the window drops below 660 pixels. This is the width at which the controls and labels on the Customers form start to wrap and become difficult to use.

3. After the trigger definition, add the following code shown in bold to the XAML markup:

```
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
        <VisualState x:Name="TabularLayout">
            <VisualState.StateTriggers>
                <AdaptiveTrigger MinWindowWidth="660"/>
            </VisualState.StateTriggers>
            <VisualState.Setters>
                <Setter Target="customersTabularView.Visibility" Value="Visible"/>
                <Setter Target="customersColumnarView.Visibility" Value="Collapsed"/>
            </VisualState.Setters>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

This code specifies the actions that occur when the trigger is fired. In this example, the actions are defined by using Setter elements. A Setter element specifies a property to set and the value that the property should be set to. For this view, the Setter commands change the values of specified properties: The customersTabularView Grid control is made visible, and the customersColumnarView Grid control is collapsed (made invisible).

4. After the TabularLayout visual state definition, add the following markup, which defines the equivalent functionality to switch to the columnar view:

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup>
    <VisualState x:Name="TabularLayout">
      ...
    </VisualState>
    <VisualState x:Name="ColumnarLayout">
      <VisualState.StateTriggers>
        <AdaptiveTrigger MinWindowWidth="0"/>
      </VisualState.StateTriggers>
      <VisualState.Setters>
        <Setter Target="customersTabularView.Visibility" Value="Collapsed"/>
        <Setter Target="customersColumnarView.Visibility" Value="Visible"/>
      </VisualState.Setters>
    </VisualState>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

When the window width drops below 660 pixels, the app switches to the ColumnarLayout state; the customersTabularView Grid control is collapsed and the customersColumnarView Grid control is made visible.

5. On the **Debug** menu, select **Start Debugging**. The app starts and displays the Customers form full screen. The data is displayed using the tabular layout.
6. Resize the Customers app window to display the form in a narrow view. When the window width drops below 660 pixels, the display switches to the columnar layout.
7. Resize the Customers app window to make it wider than 660 pixels (or maximize it to full screen). The Customers form reverts to the tabular layout.
8. Return to Visual Studio and stop debugging.

Applying styles to a UI

Now that you have the mechanics of the basic layout of the app resolved, the next step is to apply some styling to make the UI look more attractive.

The controls in a UWP app have a varied range of properties that you can use to change features such as the font, color, size, and other attributes of an element. You can set these properties

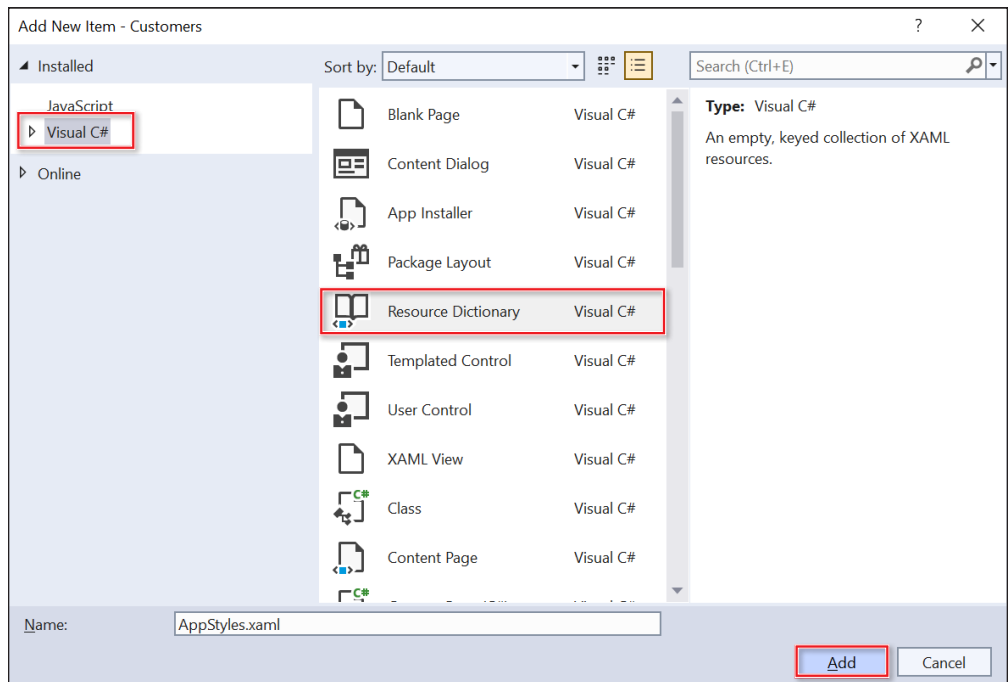
individually for each control, but this approach can become cumbersome and repetitive if your app contains lots of controls. Also, the best apps apply a consistent styling across the UI, and it's difficult to maintain consistency if you have to repeatedly set the same properties and values as you add or change controls. The more times you have to do the same thing, the greater the chances are that you will get it wrong at least once!

With UWP apps, you can define reusable styles. You can implement them as app-wide resources by creating a resource dictionary to make them available to all controls in all pages in an app. You can also define local resources that apply to only a single page in the XAML markup for that page.

In the following exercise, you'll define some simple styles for the Customers app and apply these styles to the controls on the Customers form.

To define styles for the Customers form

1. In Solution Explorer, right-click the **Customers** project, point to **Add**, and then select **New Item**.
2. In the left pane of the Add New Item – Customers dialog, select **Visual C#**. In the middle pane, select **Resource Dictionary**. In the Name box, type **AppStyles.xaml**, and then select **Add**.



The AppStyles.xaml file appears in the Code and Text Editor window. A resource dictionary is an XAML file that contains resources that the app can use. The AppStyles.xaml file looks like this:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
</ResourceDictionary>
```

Styles are one example of a resource, but you can also add other items. In fact, the first resource that you will add is not actually a style but an ImageBrush resource that will be used to paint the background of the outermost Grid control on the Customers form.

3. In Solution Explorer, right-click the **Customers** project, point to **Add**, and then select **New Folder**. Then change the name of the new folder to **Images**.
4. Right-click the **Images** folder, point to **Add**, and then select **Existing Item**.
5. In the Add Existing Item – Customers dialog, browse to the **\Microsoft Press\VCSBS\Chapter 25\Resources** folder in your **Documents** folder, select **wood.jpg**, and then select **Add**.

The wood.jpg file is added to the Images folder in the Customers project. This file contains an image of a tasteful wooden background that you'll use for the Customers form.

6. In the Code and Text Editor window displaying the AppStyles.xaml file, add the following XAML markup shown in bold:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <ImageBrush x:Key="WoodBrush" ImageSource="Images/wood.jpg"/>
</ResourceDictionary>
```

This markup creates an ImageBrush resource called WoodBrush that's based on the wood.jpg file. You can use this image brush to set the background of a control, and it will display the image in the wood.jpg file.

7. Underneath the ImageBrush resource, add the following style shown in bold to the AppStyles.xaml file:

```
<ResourceDictionary
    ...>

    <ImageBrush x:Key="WoodBrush" ImageSource="Images/wood.jpg"/>
    <Style x:Key="PageStyle" TargetType="Page">
        <Setter Property="Background" Value="{StaticResource WoodBrush}"/>
    </Style>
</ResourceDictionary>
```

This markup shows how to define a style. A `Style` element should have a name (a key that enables it to be referenced elsewhere in the app), and it should specify the type of control that the style can be applied to. You'll use this style with the `Page` control.

The body of a style consists of one or more `Setter` elements. In this example, the `Background` property is set to the `WoodBrush` `ImageBrush` resource. The syntax is a little curious, though. In a value, you can either reference one of the appropriate system-defined values for the property (such as `Red` if you want to set the background to a solid red color) or specify a resource that you have defined elsewhere. To reference a resource defined elsewhere, you use the `StaticResource` keyword and then place the entire expression in curly braces.

Before you can use this style, you must update the global resource dictionary for the app in the `App.xaml` file by adding a reference to the `AppStyles.xaml` file.

8. In Solution Explorer, double-click **App.xaml** to display it in the Code and Text Editor window. The `App.xaml` file looks like this:

```
<Application
  x:Class="Customers.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:Customers">

</Application>
```

Currently, the `App.xaml` file defines only the app object and brings a few namespaces into scope. The global resource dictionary is empty.

9. Add to the `App.xaml` file the code shown here in bold:

```
<Application
  x:Class="Customers.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:Customers" >

  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="AppStyles.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

This markup adds the resources defined in the `AppStyles.xaml` file to the list of resources available in the global resource dictionary. These resources are now available for use throughout the app.

10. Switch to the `MainPage.xaml` file displaying the UI for the Customers form. In the XAML pane, examine the markup for the Page control:

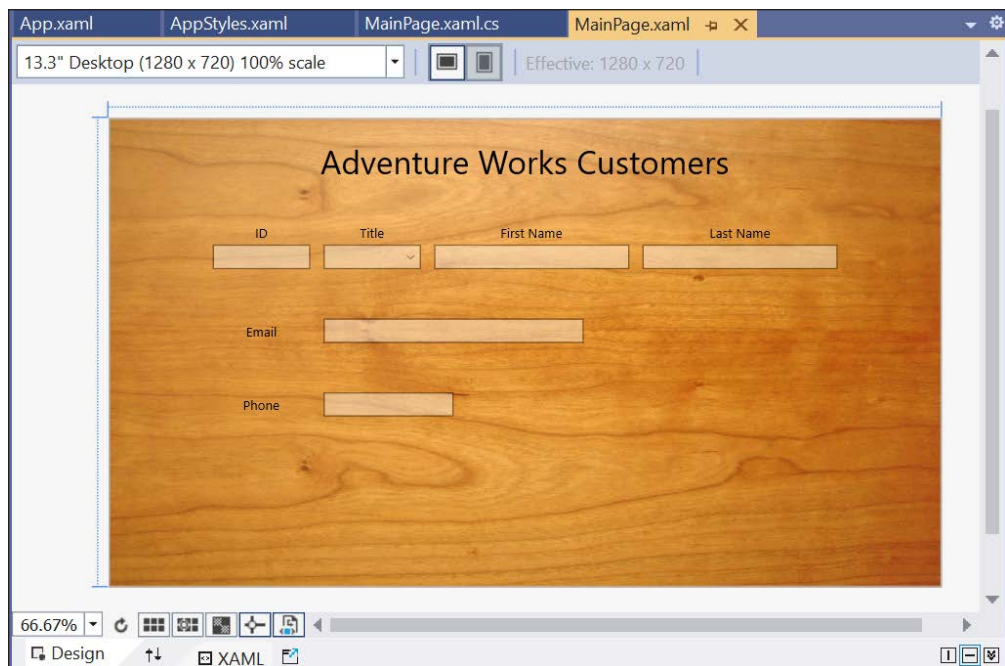
```
<Page
    x:Class="Customers.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Customers"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
```

11. In the XAML markup for this control, replace the `Background` property with a `Style` property that references the `PageStyle` style, as shown in bold in the following code:

```
<Page
    x:Class="Customers.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Customers"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Style="{StaticResource PageStyle}">
```

12. On the **Build** menu, select **Rebuild Solution**.

The background of the `Grid` control in the Design View window should switch and display a wooden panel, like this:





Note Ideally, you should ensure that any background image that you apply to a page or control maintains its aesthetics as the device form factor and orientation change. An image that looks cool on a 30-inch monitor might appear distorted and squashed on a tablet. It might be necessary to provide alternative backgrounds for different views and orientations and use the Visual State Manager to modify the Background property of a control to switch between them as the visual state changes.

13. Return to AppStyles.xaml in the Code and Text Editor window and add the following `FontStyle` style after the `PageStyle` style:

```
<Style x:Key="PageStyle" TargetType="Page">
    ...
</Style>
<Style x:Key="FontStyle" TargetType="TextBlock">
    <Setter Property="FontFamily" Value="Segoe Print"/>
</Style>
```

This style applies to `TextBlock` elements and changes the font to Segoe Print. This font resembles a handwriting style.

At this stage, it would be possible to reference the `FontStyle` style in every `TextBlock` control that required this font, but this approach wouldn't provide any advantage over simply setting the `FontFamily` directly in the markup for each control. The real power of styles occurs when you combine multiple properties, as you'll see in the next few steps.

14. Add the `HeaderStyle` style shown here in bold to the AppStyles.xaml file:

```
<Style x:Key="FontStyle" TargetType="TextBlock">
    ...
</Style>
<Style x:Key="HeaderStyle" TargetType="TextBlock" BasedOn="{StaticResource FontStyle}">
    <Setter Property="HorizontalAlignment" Value="Center"/>
    <Setter Property="TextWrapping" Value="Wrap"/>
    <Setter Property="VerticalAlignment" Value="Center"/>
    <Setter Property="Foreground" Value="SteelBlue"/>
</Style>
```

This is a composite style that sets the `HorizontalAlignment`, `TextWrapping`, `VerticalAlignment`, and `Foreground` properties of a `TextBlock` object. Additionally, the `HeaderStyle` style references the `FontStyle` style by using the `BasedOn` property. The `BasedOn` property provides a simple form of inheritance for styles.

You'll use this style to format the labels that appear at the top of the `customersTabularView` and `customersColumnarView` controls. However, these headings have different font sizes (the heading for the tabular layout is bigger than that of the columnar layout), so you'll create two more styles that extend the `HeaderStyle` style.

15. Add the styles shown here in bold to the AppStyles.xaml file:

```
<Style x:Key="HeaderStyle" TargetType="TextBlock" BasedOn="{StaticResource FontStyle}">
    ...
</Style>
<Style x:Key="TabularHeaderStyle" TargetType="TextBlock"
BasedOn="{StaticResource HeaderStyle}">
    <Setter Property="FontSize" Value="40"/>
</Style>
<Style x:Key="ColumnarHeaderStyle" TargetType="TextBlock"
BasedOn="{StaticResource HeaderStyle}">
    <Setter Property="FontSize" Value="30"/>
</Style>
```

Note that the font sizes for these styles are slightly smaller than the font sizes currently used by the headings in the Grid controls. This is because the Segoe Print font is bigger than the default font.

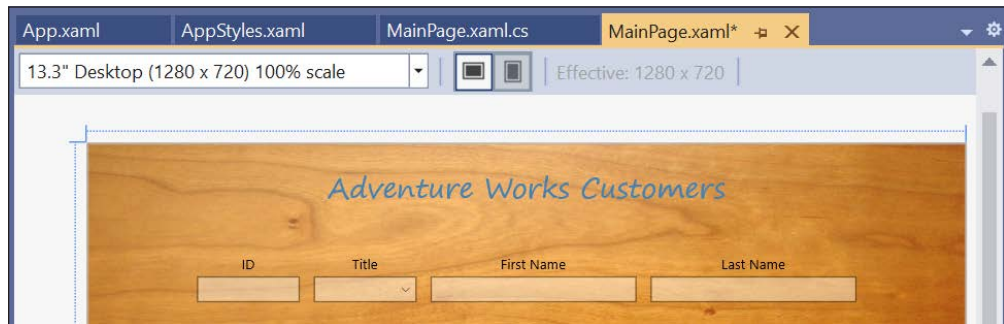
16. Switch back to the MainPage.xaml file and find the XAML markup for the TextBlock control for the Adventure Works Customers label in the customersTabularGridView Grid control:

```
<TextBlock Grid.Row="0" HorizontalAlignment="Center" TextWrapping="Wrap"
Text="Adventure Works Customers" VerticalAlignment="Center" FontSize="50"/>
```

17. Change the properties of this control to reference the TabularHeaderStyle style, as shown in bold in the following code:

```
<TextBlock Grid.Row="0" Style="{StaticResource TabularHeaderStyle}"
Text="Adventure Works Customers"/>
```

The heading displayed in the Design View window should change color, size, and font and look like this:



18. Find the XAML markup for the TextBlock control for the Customers label in the customersColumnarView Grid control:

```
<TextBlock Grid.Row="0" HorizontalAlignment="Center" TextWrapping="Wrap"
Text="Customers" VerticalAlignment="Center" FontSize="30"/>
```

Modify the markup of this control to reference the ColumnHeaderStyle style, as shown here in bold:

```
<TextBlock Grid.Row="0" Style="{StaticResource ColumnHeaderStyle}"
Text="Customers"/>
```

Be aware that you won't see this change in the Design View window because the customersColumnarView Grid control is collapsed by default. However, you will see the effects of this change when you run the app later in this exercise.

19. Return to the AppStyles.xaml file in the Code and Text Editor window. Modify the HeaderStyle style with the additional property Setter elements shown in bold in the following example:

```
<Style x:Key="HeaderStyle" TargetType="TextBlock" BasedOn="{StaticResource FontStyle}">
    <Setter Property="HorizontalAlignment" Value="Center"/>
    <Setter Property="TextWrapping" Value="Wrap"/>
    <Setter Property="VerticalAlignment" Value="Center"/>
    <Setter Property="Foreground" Value="SteelBlue"/>
    <Setter Property="RenderTransformOrigin" Value="0.5,0.5"/>
    <Setter Property="RenderTransform">
        <Setter.Value>
            <CompositeTransform Rotation="-5"/>
        </Setter.Value>
    </Setter>
</Style>
```

These elements rotate the text displayed in the header about its midpoint by an angle of 5 degrees by using a transformation.

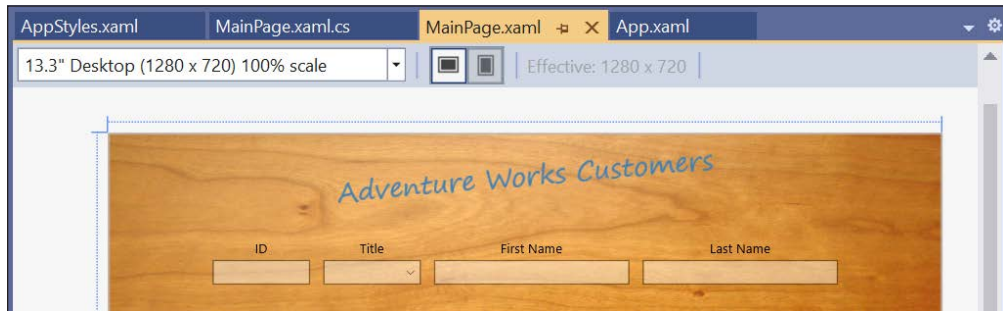


Note This example shows a simple transformation. Using the RenderTransform property, you can perform a variety of other transformations to an item, and you can combine multiple transformations. For example, you can translate (move) an item on the x- and y-axes, skew the item (make it lean), and scale an element.

You should also notice that the value of the RenderTransform property is itself another property/value pair (the property is Rotation, and the value is -5). In cases such as this, you specify the value by using the <Setter.Value> tag.

20. Rebuild the solution.

21. Switch to the MainPage.xaml file. In the Design View window, the title should now be displayed at a jaunty angle. (You might need to rebuild the application first before the updated style is applied.)



22. In the AppStyles.xaml file, add the following style:

```
<Style x:Key="LabelStyle" TargetType="TextBlock" BasedOn="{StaticResource FontStyle}">
  <Setter Property="FontSize" Value="20"/>
  <Setter Property="HorizontalAlignment" Value="Center"/>
  <Setter Property="TextWrapping" Value="Wrap"/>
  <Setter Property="VerticalAlignment" Value="Center"/>
  <Setter Property="Foreground" Value="AntiqueWhite"/>
</Style>
```

You'll apply this style to the TextBlock controls that provide the labels for the various TextBox and ComboBox controls that the user employs to enter customer information. The style references the same font style as the headings but sets the other properties to values more appropriate for the labels.

23. Go back to the MainPage.xaml file. In the **XAML** pane, modify the markup for the TextBlock controls for each of the labels in the customersTabularView and customersColumnarView Grid controls. Remove the HorizontalAlignment, TextWrapping, VerticalAlignment, and FontSize properties, and reference the LabelStyle style, as shown here in bold:

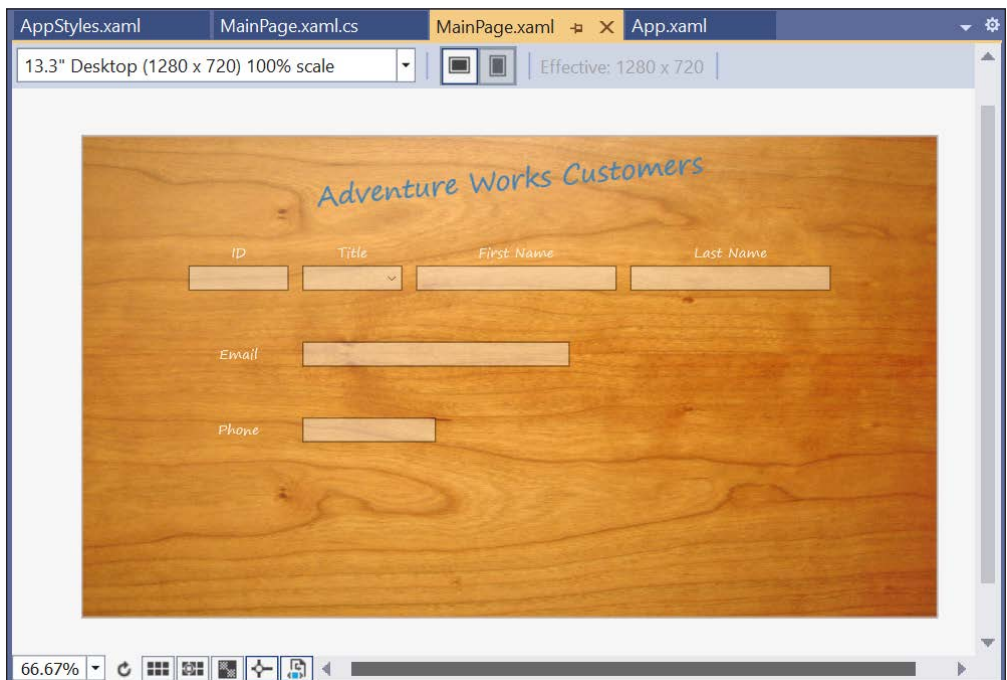
```
<Grid x:Name="customersTabularView" Margin="10,20,10,20" Visibility="Visible">
  ...
  <Grid Grid.Row="2">
    ...
    <TextBlock Grid.Row="0" Grid.Column="1" Style="{StaticResource LabelStyle}"
Text="ID"/>
    <TextBlock Grid.Row="0" Grid.Column="3" Style="{StaticResource LabelStyle}"
Text="Title"/>
    <TextBlock Grid.Row="0" Grid.Column="5" Style="{StaticResource LabelStyle}"
Text="First Name"/>
    <TextBlock Grid.Row="0" Grid.Column="7" Style="{StaticResource LabelStyle}"
Text="Last Name"/>
    ...
    <TextBlock Grid.Row="3" Grid.Column="1" Style="{StaticResource LabelStyle}"
Text="Email"/>
    ...
  </Grid>
</Grid>
```

```

        <TextBlock Grid.Row="5" Grid.Column="1" Style="{StaticResource LabelStyle}"
Text="Phone"/>
        ...
    </Grid>
</Grid>
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Collapsed">
    ...
    <Grid Grid.Row="1">
        ...
        <TextBlock Grid.Row="0" Grid.Column="0" Style="{StaticResource LabelStyle}"
Text="ID"/>
        <TextBlock Grid.Row="1" Grid.Column="0" Style="{StaticResource LabelStyle}"
Text="Title"/>
        <TextBlock Grid.Row="2" Grid.Column="0" Style="{StaticResource LabelStyle}"
Text="First Name"/>
        <TextBlock Grid.Row="3" Grid.Column="0" Style="{StaticResource LabelStyle}"
Text="Last Name"/>
        ...
        <TextBlock Grid.Row="4" Grid.Column="0" Style="{StaticResource LabelStyle}"
Text="Email"/>
        ...
        <TextBlock Grid.Row="5" Grid.Column="0" Style="{StaticResource LabelStyle}"
Text="Phone"/>
        ...
    </Grid>
</Grid>

```

The labels on the form should change to the Segoe Print font and be displayed in white, in a font size of 30 pixels:



24. On the **Debug** menu, select **Start Debugging** to build and run the app.

The Customers form should appear and be styled in the same way that it appears in the Design View window in Visual Studio. Notice that if you enter any text into the various fields on the form, they use the default font and styling for the `TextBox` controls.



Note Although the Segoe Print font is good for labels and titles, it is not recommended as a font for data-entry fields because some of the characters can be difficult to distinguish from one another. For example, the lowercase letter `l` is very similar to the digit `1`, and the uppercase letter `O` is almost indistinguishable from the digit `0`. For this reason, it makes sense to stick with the default font for the `TextBox` controls.

25. Resize the window to make it narrower and verify that the styling has been applied to the controls in the `customersColumnarView` grid. The form should look like the following image:



Note If you've entered any text into the form in wide view, this text is not currently transferred to the narrow view. You'll see why this is the case, and fix it, in the next chapter.

26. Return to Visual Studio and stop debugging.

You can see that by using styles, you can easily implement a variety of really cool effects. Also, careful use of styles makes your code much more maintainable than it would be if you set properties on individual controls. For example, if you want to switch the font used by the labels and headings in the Customers app, you need to make only a single change to the `FontStyle` style.

In general, you should use styles wherever possible. In addition to assisting with maintainability, the use of styles helps to keep the XAML markup for your forms clean and uncluttered, and the XAML for a form needs to specify only the controls and layout rather than how the controls should appear on the form.



Tip You can use Microsoft Blend for Visual Studio 2022 to define complex styles that you can integrate into an app. Professional graphic artists frequently use Blend to develop custom styles and provide these styles in the form of XAML markup to developers building apps. All the developer has to do is add the appropriate `Style` tags to the user interface elements to reference the corresponding styles.

Summary

In this chapter, you learned how to use the `Grid` control to implement a user interface that can scale to different device form factors and orientations. You also learned how to use the Visual State Manager to adapt the layout of controls when the user changes the size of the window displaying the app. Finally, you learned how to create custom styles and apply them to the controls on a form. Now that you've defined the user interface, the next challenge is to add functionality to the app, enabling the user to display and update data, which is what you'll do in the final chapters.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 26.
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

To	Do this
Create a new UWP app	Use one of the UWP templates in Visual Studio 2022, such as the Blank App template.
Implement a user interface that scales to different device form factors and orientations	Use a Grid control. Divide the Grid control into rows and columns, and place controls in these rows and columns rather than specifying an absolute location relative to the edges of the Grid.
Implement a user interface that can adapt to different display widths	Create different layouts for each view that display the controls in an appropriate manner. Use the Visual State Manager to select the layout to display when the visual state changes.
Create custom styles	Add a resource dictionary to the app. Define styles in this dictionary by using the <Style> element and specify the properties that each style changes. For example: <code><Style x:Key="PageStyle" TargetType="Page"> <Setter Property="Background" Value="{StaticResource WoodBrush}"/> </Style></code>
Apply a custom style to a control	Set the Style property of the control and reference the style by name. For example: <code><Grid Style="{StaticResource PageStyle}"></code>

Displaying and searching for data in a Universal Windows Platform app

After completing this chapter, you will be able to:

- Explain how to use the Model-View-ViewModel pattern to implement the logic for a Universal Windows Platform app.
- Use data binding to display and modify data in a view.
- Create a ViewModel with which a view can interact with a model.

Chapter 25, “Implementing the user interface for a Universal Windows Platform app,” demonstrated how to design a user interface (UI) that can adapt to the different device form factors, orientations, and views that a user running your app might employ. The sample app developed in that chapter is a simple one designed for displaying and editing details about customers.

In this chapter, you’ll see how to display data in the UI and learn about the features in Windows 11 that you can use to search for data in an app. In performing these tasks, you’ll also learn about the best way to structure a Universal Windows Platform (UWP) app. This chapter covers a lot of ground. In particular, you’ll look at how to use data binding to connect the UI to the data that it displays and how to create a ViewModel to separate the user interface logic from the data model and business logic for an app.

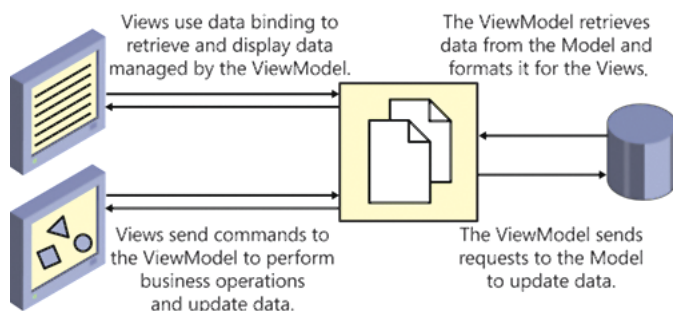
Implementing the Model-View-ViewModel pattern

A well-structured graphical app separates the design of the UI from the data that the application uses and the business logic that comprises the functionality of the app. This separation helps to remove the dependencies between the various components, enabling different presentations of the data without needing to change the business logic or the underlying data model. This approach also clears the way for different elements to be designed and implemented by individuals who have the appropriate specialist skills. For example, a graphic artist can focus on designing an appealing and intuitive UI,

a database specialist can concentrate on implementing an optimized set of data structures for storing and accessing the data, and a C# developer can direct her efforts toward implementing the business logic for the app. This common goal has been the aim of many development approaches—not just for UWP apps—and many techniques have been devised over the past few years to help structure an app in this way.

Arguably, the most popular approach is to follow the Model-View-ViewModel (MVVM) design pattern. In this design pattern, the model provides the data used by the app, and the view represents the way in which the data is displayed in the UI. The ViewModel contains the logic that connects the two, converting the user input into commands that perform business operations on the model, and also formatting the data from the model in the manner expected by the view.

The following diagram shows a simplified relationship between the elements of the MVVM pattern. Note that an app might provide multiple views of the same data. In a UWP app, for example, you might implement different view states, which can present information by using different screen layouts. One job of the ViewModel is to ensure that the data from the same model can be displayed and manipulated by many different views. In a UWP app, the view can use data binding to connect to the data presented by the ViewModel. Additionally, the view can request that the ViewModel update data in the model or perform business tasks by invoking commands implemented by the ViewModel.



Displaying data by using data binding

Before you get started implementing a ViewModel for the Customers app, it helps to understand a little more about data binding and how you can apply it to display data in a UI. Using data binding, you can link a property of a control to a property of an object; then, if the value of the specified property of the object changes, the property in the control that's linked to the object also changes.

Also, data binding can be bidirectional. If the value of a property in a control that uses data binding changes, the modification is propagated to the object the control is linked to. The following exercise provides a quick introduction to how data binding is used to display data. It's based on the Customers app from Chapter 25.

To use data binding to display Customer information

1. Start Visual Studio 2022, if it is not already running.
2. Open the **Customers** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 26\DataBinding** folder in your **Documents** folder. This is a version of the Customers app developed in Chapter 25, but the layout of the UI has been modified slightly; the controls are displayed on a blue background, which makes them stand out more easily.



Note The blue background was created by using a `Rectangle` control that spans the same rows and columns as the `TextBlock` and `TextBox` controls that display the headings and data. The rectangle is filled by using a `LinearGradientBrush` that gradually changes the color of the rectangle from a medium blue at the top to a very dark blue at the bottom. The XAML markup for the `Rectangle` control that's displayed in the `customersTabularViewGrid` control looks like this. (The XAML markup for the `customersColumnarViewGrid` control includes a similar `Rectangle` control, spanning the rows and columns used by that layout.)

```
<Rectangle Grid.Row="0" Grid.RowSpan="6" Grid.Column="1" Grid.ColumnSpan="7" ...>
  <Rectangle.Fill>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="#FF0E3895"/>
      <GradientStop Color="#FF141415" Offset="0.929"/>
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
```

3. In Solution Explorer, right-click the **Customers** project, point to **Add**, and then select **Class**.
4. In the Add New Items – Customers dialog, ensure that the **Class** template is selected. In the **Name** box, type **Customer.cs**. Then select **Add**.

You'll use this class to implement the Customer data type and then implement data binding to display the details of Customer objects in the UI.

5. In the Code and Text Editor window displaying the Customer.cs file, make the Customer class public and add the following private fields and public properties shown in bold:

```
public class Customer
{
    private int _customerID;
    public int CustomerID
    {
        get => this._customerID;
        set
        {
            this._customerID = value;
        }
    }
}
```

```

private string _title;
public string Title
{
    get => this._title;
    set
    {
        this._title = value;
    }
}

private string _firstName;
public string FirstName
{
    get => this._firstName;
    set
    {
        this._firstName = value;
    }
}

private string _lastName;
public string LastName
{
    get => this._lastName;
    set
    {
        this._lastName = value;
    }
}

private string _emailAddress;
public string EmailAddress
{
    get => this._emailAddress;
    set
    {
        this._emailAddress = value;
    }
}

private string _phone;
public string Phone
{
    get => this._phone;
    set
    {
        this._phone = value;
    }
}
}

```

You might be wondering why the class doesn't simply define automatic properties, or why the property setters are not implemented as expression-bodied members, given that all they do is set the value in a private field. The answer to both of these questions is that you will add code to these properties in a later exercise.

6. In Solution Explorer, in the Customers project, double-click the **MainPage.xaml** file to display the user interface for the application in the Design View window.
7. In the **XAML** pane, locate the markup for the `id` TextBox control. Modify the XAML markup that sets the `Text` property for this control as shown here in bold:

```
<TextBox Grid.Row="1" Grid.Column="1" x:Name="id" ...
Text="{Binding CustomerID}" .../>
```

The syntax `Text="{Binding Path}"` specifies that the value of the `Text` property will be provided by the value of the `Path` expression at runtime. In this case, `Path` is set to `CustomerID`, so the value held in the `CustomerID` expression will be displayed by this control. However, you need to provide a bit more information to indicate that `CustomerID` is actually a property of a `Customer` object. To do this, you set the `DataContext` property of the control, which you'll do shortly.

8. Add the following binding expressions for each of the other text controls on the form. Apply data binding to the TextBox controls in the `customersTabularView` and `customersColumnarView` Grid controls, as shown in bold in the following code. (The `ComboBox` controls require slightly different handling, which you'll address in the section "Using data binding with a `ComboBox` control" later in this chapter.)

```
<Grid x:Name="customersTabularView" ...>
    ...
    <TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName" ...
Text="{Binding FirstName}" .../>
    <TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName" ...
Text="{Binding LastName}" .../>
    ...
    <TextBox Grid.Row="3" Grid.Column="3" Grid.ColumnSpan="5"
x:Name="email" ... Text="{Binding EmailAddress}" .../>
    ...
    <TextBox Grid.Row="5" Grid.Column="3" Grid.ColumnSpan="3"
x:Name="phone" ... Text="{Binding Phone}" .../>
</Grid>
<Grid x:Name="customersColumnarView" Margin="10,20,10,20"
Visibility="Collapsed">
    ...
    <TextBox Grid.Row="0" Grid.Column="1" x:Name="cId" ...
Text="{Binding CustomerID}" .../>
    ...
    <TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName" ...
Text="{Binding FirstName}" .../>
    <TextBox Grid.Row="3" Grid.Column="1" x:Name="cLastName" ...
Text="{Binding LastName}" .../>
    ...
    <TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail" ...
Text="{Binding EmailAddress}" .../>
    ...
    <TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone" ...
Text="{Binding Phone}" .../>
</Grid>
```


Notice how the same binding expression can be used with more than one control. For example, the expression `{Binding CustomerID}` is referenced by the `id` and `cId` `TextBox` controls, which causes both controls to display the same data.

9. In Solution Explorer, expand the **MainPage.xaml** file, and then double-click the **MainPage.xaml.cs** file to display the code for the `MainPage.xaml` form in the Code and Text Editor window.
10. Add the statement shown here in bold to the `MainPage` constructor:

```
public MainPage()
{
    this.InitializeComponent();
    Customer customer = new Customer
    {
        CustomerID = 1,
        Title = "Mr",
        FirstName = "John",
        LastName = "Sharp",
        EmailAddress = "john@contoso.com",
        Phone = "111-1111"
    };
}
```

This code creates a new instance of the `Customer` class and populates it with some sample data.

11. After the code that creates the new `Customer` object, add the following statement shown in bold:

```
Customer customer = new Customer
{
    ...
};
this.DataContext = customer;
```

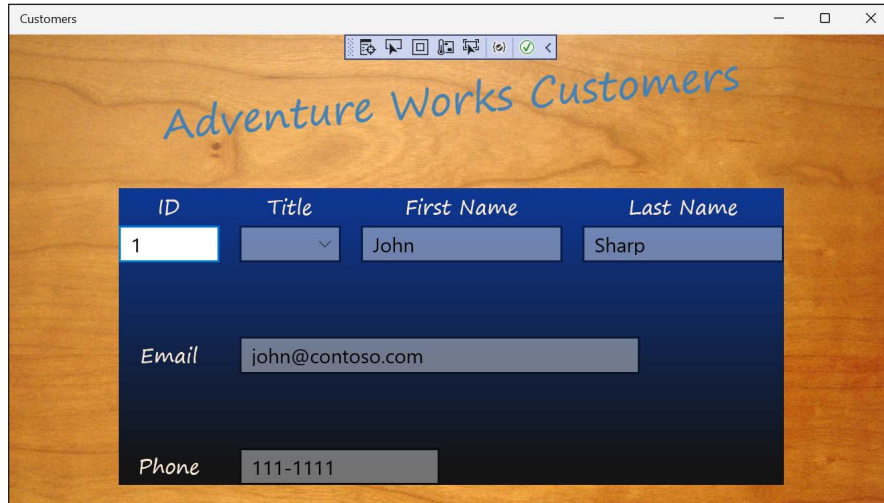
This statement specifies the object to which controls on the `MainPage` form should bind. In each of the controls, the XAML markup `Text="{Binding Path}"` will be resolved against this object. For example, the `id` `TextBox` and `cId` `TextBox` controls both specify `Text="{Binding CustomerID}"`, so they will display the value found in the `CustomerID` property of the `Customer` object that the form is bound to.



Note In this example, you've set the `DataContext` property of the form, so the same data binding automatically applies to all the controls on the form. You can also set the `DataContext` property for individual controls if you need to bind specific controls to different objects.

12. On the **Debug** menu, select **Start Debugging** to build and run the app.

Verify that the form occupies the full screen and displays the details for the customer John Sharp, as shown in the following image:



The screenshot shows a window titled 'Customers' with a wooden background. At the top, the text 'Adventure Works Customers' is written in a blue, handwritten-style font. Below this, there is a form with a dark blue background. The form contains the following fields:

ID	Title	First Name	Last Name
1	▼	John	Sharp

Below the table, there are two more fields:

Email: john@contoso.com

Phone: 111-1111

13. Resize the app window to display it in the narrow view. Verify that it displays the same data, as illustrated here:



The screenshot shows the same 'Customers' app window, but now in a narrow view. The title bar still says 'Customers'. The background is still wooden, but the text 'Customers' is written in a blue, handwritten-style font. The form has a dark blue background and contains the following fields:

ID: 1

Title: ▼

First Name: John

Last Name: Sharp

Email: john@contoso.com

Phone: 111-1111

The controls displayed in the narrow view are bound to the same data as the controls displayed in the full-screen view.

14. In the narrow view, change the email address to **john@tresearch.com**.
15. Expand the app window to switch to the wide view. Notice that the email address displayed in this view has not changed.
16. Return to Visual Studio and stop debugging.
17. In Visual Studio, display the code for the `Customer` class in the Code and Text Editor window and set a breakpoint in the `set` property accessor for the `EmailAddress` property.
18. On the **Debug** menu, select **Start Debugging** to build and run the application again.
19. When the debugger reaches the breakpoint for the first time, press **F5** to continue running the app.
20. When the UI for the Customers app appears, resize the application window to display the narrow view and change the email address to **john@tresearch.com**.
21. Expand the app window back to the wide view.

The debugger doesn't reach the breakpoint in the `set` accessor for the `EmailAddress` property. The updated value is not written back to the `Customer` object when the email `TextBox` loses the focus.

22. Return to Visual Studio and stop debugging.
23. Remove the breakpoint in the `set` accessor of the `EmailAddress` property in the `Customer` class.

Modifying data by using data binding

In the previous exercise, you saw how easily data in an object can be displayed by using data binding. However, data binding is a one-way operation by default, and any changes you make to the displayed data are not copied back to the data source. In the exercise, you saw this when you changed the email address displayed in the narrow view; when you switched back to the wide view, the data had not changed.

You can implement bidirectional data binding by modifying the `Mode` parameter of the `Binding` specification in the XAML markup for a control. The `Mode` parameter indicates whether data binding is one-way or two-way. This is what you'll do next.

To implement two-way data binding to modify customer information

1. Display the MainPage.xaml file in the Design View window and modify the XAML markup for each of the TextBox controls as shown in bold in the following code:

```
<Grid x:Name="customersTabularView" ...>
    ...
    <TextBox Grid.Row="1" Grid.Column="1" x:Name="id" ...
    Text="{Binding CustomerID, Mode=TwoWay}" .../>
    ...
    <TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName" ...
    Text="{Binding FirstName, Mode=TwoWay}" .../>
    <TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName" ...
    Text="{Binding LastName, Mode=TwoWay}" .../>
    ...
    <TextBox Grid.Row="3" Grid.Column="3" Grid.ColumnSpan="5"
    x:Name="email" ... Text="{Binding EmailAddress, Mode=TwoWay}" .../>
    ...
    <TextBox Grid.Row="5" Grid.Column="3" Grid.ColumnSpan="3"
    x:Name="phone" ... Text="{Binding Phone, Mode=TwoWay}" .../>
</Grid>
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" ...>
    ...
    <TextBox Grid.Row="0" Grid.Column="1" x:Name="cId" ...
    Text="{Binding CustomerID, Mode=TwoWay}" .../>
    ...
    <TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName" ...
    Text="{Binding FirstName, Mode=TwoWay}" .../>
    <TextBox Grid.Row="3" Grid.Column="1" x:Name="cLastName" ...
    Text="{Binding LastName, Mode=TwoWay}" .../>
    ...
    <TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail" ...
    Text="{Binding EmailAddress, Mode=TwoWay}" .../>
    ...
    <TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone" ...
    Text="{Binding Phone, Mode=TwoWay}" .../>
</Grid>
```

The Mode parameter to the Binding specification indicates whether data binding is one-way (the default) or two-way. Setting Mode to TwoWay causes any changes made by the user to be passed back to the object that a control is bound to.

2. On the **Debug** menu, select **Start Debugging** to build and run the app again.
3. With the app in the wide view, change the email address to **john@treystudy.com**, and then resize the window to display the app in the narrow view.

Despite the change in the data binding to two-way mode, the email address displayed in the narrow view hasn't been updated; it's still *john@contoso.com*.

4. Return to Visual Studio and stop debugging.

Clearly, something is not working correctly! The problem now is not that the data hasn't been updated but rather that the view isn't displaying the latest version of the data. (If you reinstate the breakpoint in the set accessor for the `EmailAddress` property of the `Customer` class and run the app in the debugger, you'll see the debugger reach the breakpoint whenever you change the value of the email address and move the focus away from the `TextBox` control.) Despite appearances, the data-binding process is not magic, and a data binding doesn't know when the data to which it is bound has been changed. The object needs to inform the data binding of any modifications by sending a `PropertyChanged` event to the UI. This event is part of an interface named `INotifyPropertyChanged`, and all objects that support two-way data binding should implement this interface. You'll implement this interface in the next exercise.

To implement the `INotifyPropertyChanged` interface in the `Customer` class

1. In Visual Studio, display the `Customer.cs` file in the Code and Text Editor window.
2. Add the following `using` directive to the list at the top of the file:

```
using System.ComponentModel;
```

The `INotifyPropertyChanged` interface is defined in this namespace.

3. Modify the definition of the `Customer` class to specify that it implements the `INotifyPropertyChanged` interface, as shown here in bold:

```
public class Customer : INotifyPropertyChanged
{
    ...
}
```

4. After the `Phone` property at the end of the `Customer` class, add the `PropertyChanged` event shown in bold in the following code:

```
public class Customer : INotifyPropertyChanged
{
    ...
    private string _phone;
    public string Phone
    {
        get => this._phone;
        set { this._phone = value; }
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```

This event is the only item that the `INotifyPropertyChanged` interface defines. All objects that implement this interface must provide this event, and they should raise this event whenever they want to notify the outside world of a change to a property value.

5. Add the `OnPropertyChanged` method shown here in bold to the `Customer` class, after the `PropertyChanged` event:

```
class Customer : INotifyPropertyChanged
{
    ...
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged is not null)
        {
            PropertyChanged(this,
                new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

The `OnPropertyChanged` method raises the `PropertyChanged` event. The `PropertyChangedEventArgs` parameter to the `PropertyChanged` event should specify the name of the property that has changed. This value is passed in as a parameter to the `OnPropertyChanged` method.



Note You can reduce the code in the `OnPropertyChanged` method to a single statement using the `null` conditional operator (`?.`) and the `Invoke` method, like this:

```
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
```

However, my personal preference is readability over terse code because better readability makes it easier to maintain your applications.

6. Modify the property set accessors for each of the properties in the `Customer` class to call the `OnPropertyChanged` method whenever the value that they contain is modified, as shown in bold here:

```
public class Customer : INotifyPropertyChanged
{
    private int _customerID;
    public int CustomerID
    {
        get => this._customerID;
        set
        {
            this._customerID = value;
            this.OnPropertyChanged(nameof(CustomerID));
        }
    }
}
```

```

private string _title;
public string Title
{
    get => this._title;
    set
    {
        this._title = value;
        this.OnPropertyChanged(nameof(Title));
    }
}

private string _firstName;
public string FirstName
{
    get => this._firstName;
    set
    {
        this._firstName = value;
        this.OnPropertyChanged(nameof(FirstName));
    }
}

private string _lastName;
public string LastName
{
    get => this._lastName;
    set
    {
        this._lastName = value;
        this.OnPropertyChanged(nameof(LastName));
    }
}

private string _emailAddress;
public string EmailAddress
{
    get => this._emailAddress;
    set
    {
        this._emailAddress = value;
        this.OnPropertyChanged(nameof(EmailAddress));
    }
}

private string _phone;
public string Phone
{
    get => this._phone;
    set
    {
        this._phone = value;
        this.OnPropertyChanged(nameof(Phone));
    }
}
...
}

```

The nameof operator

The `nameof` operator demonstrated in the `Customer` class is a little-used but valuable feature of C# in code such as this. It returns the name of the variable passed in as its parameter as a string. Without using the `nameof` operator, you would've had to use hard-coded string values. For example:

```
public int CustomerID
{
    get { return this._customerID; }
    set
    {
        this._customerID = value;
        this.OnPropertyChanged("CustomerID");
    }
}
```

Although using the string values requires less typing, consider what would happen if you needed to change the name of the property at some point in the future. Using the string approach, you'd need to modify the string value as well. If you didn't, the code would still compile and run, but any changes made to the property value at runtime would not be notified, leading to difficult-to-find bugs. Using the `nameof` operator, if you change the name of the property but forget to change the argument to `nameof`, the code won't compile, alerting you immediately to an error that should be quick and easy to fix.

7. On the **Debug** menu, select **Start Debugging** to build and run the app again.
8. When the Customers form appears, change the email address to **john@treystudy.com**, and change the phone number to **222-2222**.
9. Resize the window to display the app in the narrow view and verify that the email address and phone number have changed.
10. Change the first name to **James**, expand the window to display the wide view, and verify that the first name has changed.
11. Return to Visual Studio and stop debugging.

Using data binding with a ComboBox control

Using data binding with a control such as a `TextBox` or `TextBlock` is relatively straightforward. On the other hand, `ComboBox` controls require a little more attention. The issue is that a `ComboBox` control actually displays two things:

- A list of values in the dropdown list that the user can select an item from
- The value of the currently selected item

If you implement data binding to display a list of items in the dropdown list of a `ComboBox` control, the value that the user selects must be a member of this list. In the `Customers` app, you can configure data binding for the selected value in the `title` `ComboBox` control by setting the `SelectedValue` property, like this:

```
<ComboBox ... x:Name="title" ... SelectedValue="{Binding Title}" ... />
```

However, remember that the list of values for the dropdown list is hard coded into the XAML markup, like this:

```
<ComboBox ... x:Name="title" ... >
    <ComboBoxItem Content="Mr"/>
    <ComboBoxItem Content="Mrs"/>
    <ComboBoxItem Content="Ms"/>
    <ComboBoxItem Content="Miss"/>
</ComboBox>
```

This markup is not applied until the control has been created, so the value specified by the data binding is not found in the list because the list does not yet exist when the data binding is constructed. The result is that the value is not displayed. You can try this if you like—configure the binding for the `SelectedValue` property as just shown and run the app. The `title` `ComboBox` will be empty when it's initially displayed, despite the fact that the customer has the title of `Mr`.

There are several solutions to this problem, but the simplest is to create a data source that contains the list of valid values and then specify that the `ComboBox` control should use this list as its set of values for the dropdown. Also, you need to do this before the data binding for the `ComboBox` control is applied.

To implement data binding for the `title` `ComboBox` controls

1. In Visual Studio, display the `MainPage.xaml.cs` file in the Code and Text Editor window.
2. Add the following code shown in bold to the `MainPage` constructor:

```
public MainPage()
{
    this.InitializeComponent();

    List<string> titles = new List<string>
    {
        "Mr", "Mrs", "Ms", "Miss"
    };
    this.title.ItemsSource = titles;
    this.cTitle.ItemsSource = titles;

    Customer customer = new Customer
    {
        ...
    };
    this.DataContext = customer;
}
```

This code creates a list of strings containing the valid titles that customers can have. The code then sets the `ItemsSource` property of both `title` `ComboBox` controls to reference this list. (Remember that each view has a `ComboBox` control.)



Note In a commercial app, you would most likely retrieve the list of values displayed by a `ComboBox` control from a database or some other data source rather than a hard-coded list, as shown in this example.

The placement of this code is important. It must run before the statement that sets the `DataContext` property of the `MainPage` form because this statement is when the data binding to the controls on the form occurs.

3. Display the `MainPage.xaml` file in the Design View window.
4. Modify the XAML markup for the `title` and `cTitle` `ComboBox` controls, as shown here in bold:

```
<Grid x:Name="customersTabularView" ...>
    ...
    <ComboBox Grid.Row="1" Grid.Column="3" x:Name="title" ...
SelectedValue="{Binding Title, Mode=TwoWay}">
</ComboBox>
    ...
</Grid>
<Grid x:Name="customersColumnarView" ...>
    ...
    <ComboBox Grid.Row="1" Grid.Column="1" x:Name="cTitle" ...
SelectedValue="{Binding Title, Mode=TwoWay}">
</ComboBox>
    ...
</Grid>
```

Notice that the list of `ComboBoxItem` elements for each control has been removed and that the `SelectedValue` property is configured to use data binding with the `Title` field in the `Customer` object.

5. On the **Debug** menu, select **Start Debugging** to build and run the application.
6. Verify that the value of the customer's title is displayed correctly (it should be Mr). Select the dropdown arrow for the `ComboBox` control and verify that it contains the values Mr, Mrs, Ms, and Miss.
7. Resize the window to display the app in the narrow view and perform the same checks. Note that you can change the title, and when you switch back to the wide view, the new title is displayed.
8. Return to Visual Studio and stop debugging.

Creating a ViewModel

You've now seen how to configure data binding to connect a data source to the controls in a user interface, but the data source that you've been using is very simple, consisting of a single customer. In the real world, the data source is likely to be much more complex, comprising collections of different types of objects.

Remember that in MVVM terms, the data source is often provided by the model, and the UI (the view) communicates with the model only indirectly through a ViewModel object. The rationale behind this approach is that the model and the views that display the data provided by the model should be independent; you shouldn't have to change the model if the user interface is modified, nor should you be required to adjust the UI if the underlying model changes.

The ViewModel provides the connection between the view and the model, and it also implements the business logic for the app. Again, this business logic should be independent of the view and the model. The ViewModel exposes the business logic to the view by implementing a collection of commands. The UI can trigger these commands based on how the user navigates through the app.

In the following exercise, you'll extend the Customers app by implementing a model that contains a list of Customer objects and creating a ViewModel that provides commands with which a user can move between customers in the view.

To create a ViewModel for managing customer information

1. Open the **Customers** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 26\ViewModel** folder in your **Documents** folder. This project contains a completed version of the Customers app from the previous set of exercises; if you prefer, you can continue to use your own version of the project.
2. In Solution Explorer, right-click the **Customers** project, point to **Add**, and then select **Class**.
3. In the Add New Items – Customers dialog, in the **Name** box, type **ViewModel.cs**, and then select **Add**.

You'll use this class to provide a basic ViewModel that contains a collection of Customer objects. The user interface will bind to the data exposed by this ViewModel.

4. In the Code and Text Editor window displaying the ViewModel.cs file, mark the class as public and add the code shown in bold in the following example to the **ViewModel** class:

```
public class ViewModel
{
    private List<Customer> customers;

    public ViewModel()
    {
        this.customers = new List<Customer>
        {
```

```

        new Customer
        {
            CustomerID = 1,
            Title = "Mr",
            FirstName="John",
            LastName="Sharp",
            EmailAddress="john@contoso.com",
            Phone="111-1111"
        },
        new Customer
        {
            CustomerID = 2,
            Title = "Mrs",
            FirstName="Diana",
            LastName="Sharp",
            EmailAddress="diana@contoso.com",
            Phone="111-1112"
        },
        new Customer
        {
            CustomerID = 3,
            Title = "Ms",
            FirstName="Francesca",
            LastName="Sharp",
            EmailAddress="frankie@contoso.com",
            Phone="111-1113"
        }
    };
}

```

The `ViewModel` class uses a `List<Customer>` object as its model, and the constructor populates this list with some sample data. Strictly speaking, this data should be held in a separate `Model` class, but for the purposes of this exercise we'll make do with this sample data.

5. Add the private variable `currentCustomer` shown in bold in the following code to the `ViewModel` class, and initialize this variable to zero in the constructor:

```

class ViewModel
{
    private List<Customer> customers;
    private int currentCustomer;

    public ViewModel()
    {
        this.currentCustomer = 0;
        this.customers = new List<Customer>
        {
            ...
        }
    }
}

```

The `ViewModel` class will use this variable to track which `Customer` object the view is currently displaying.

6. Add the **Current** property shown here in bold to the `ViewModel` class, after the constructor:

```
class ViewModel
{
    ...
    public ViewModel()
    {
        ...
    }

    public Customer Current
    {
        get => this.customers.Count > 0 ? this.customers[currentCustomer] : null;
    }
}
```

The **Current** property provides access to the current `Customer` object in the model. If there are no customers, it returns a null object.



Note It's good practice to provide controlled access to a data model; only the `ViewModel` should be able to modify the model. However, this restriction doesn't prevent the view from being able to update the data presented by the `ViewModel`; it just can't switch the model and make it refer to a different data source.

7. Open the `MainPage.xaml.cs` file in the Code and Text Editor window.
8. In the `MainPage` constructor, remove the code that creates the `Customer` object and replace it with a statement that creates an instance of the `ViewModel` class. Change the statement that sets the `DataContext` property of the `MainPage` object to reference the new `ViewModel` object, as shown here in bold:

```
public MainPage()
{
    ...
    this.cTitle.ItemsSource = titles;
    ViewModel = new ViewModel();
    this.DataContext = viewModel;
}
```

9. Open the `MainPage.xaml` file in the Design View window.
10. In the **XAML** pane, modify the data bindings for the `TextBox` and `ComboBox` controls to reference properties through the `Current` object presented by the `ViewModel`, as shown in bold in the following code:

```
<Grid x:Name="customersTabularView" ...>
    ...
    <TextBox Grid.Row="1" Grid.Column="1" x:Name="id" ...
    Text="{Binding Current.CustomerID, Mode=TwoWay}" .../>
    <TextBox Grid.Row="1" Grid.Column="5" x:Name="firstName" ...
    Text="{Binding Current.FirstName, Mode=TwoWay }" .../>
```

```

        <TextBox Grid.Row="1" Grid.Column="7" x:Name="lastName" ...
Text="{Binding Current.LastName, Mode=TwoWay }" .../>
        <ComboBox Grid.Row="1" Grid.Column="3" x:Name="title" ...
SelectedValue="{Binding Current.Title, Mode=TwoWay}">
        </ComboBox>
        ...
        <TextBox Grid.Row="3" Grid.Column="3" ... x:Name="email" ...
Text="{Binding Current.EmailAddress, Mode=TwoWay }" .../>
        ...
        <TextBox Grid.Row="5" Grid.Column="3" ... x:Name="phone" ...
Text="{Binding Current.Phone, Mode=TwoWay }" .../>
    </Grid>
    <Grid x:Name="customersColumnarView" Margin="20,10,20,110" ...>
        ...
        <TextBox Grid.Row="0" Grid.Column="1" x:Name="cId" ...
Text="{Binding Current.CustomerID, Mode=TwoWay }" .../>
        <TextBox Grid.Row="2" Grid.Column="1" x:Name="cFirstName" ...
Text="{Binding Current.FirstName, Mode=TwoWay }" .../>
        <TextBox Grid.Row="3" Grid.Column="1" x:Name="cLastName" ...
Text="{Binding Current.LastName, Mode=TwoWay }" .../>
        <ComboBox Grid.Row="1" Grid.Column="1" x:Name="cTitle" ...
SelectedValue="{Binding Current.Title, Mode=TwoWay}">
        </ComboBox>
        ...
        <TextBox Grid.Row="4" Grid.Column="1" x:Name="cEmail" ...
Text="{Binding Current.EmailAddress, Mode=TwoWay }" .../>
        ...
        <TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone" ...
Text="{Binding Current.Phone, Mode=TwoWay }" .../>
    </Grid>

```

11. On the **Debug** menu, select **Start Debugging** to build and run the app.
12. Verify that the app displays the details of John Sharp (the first customer in the customers list). Change the details of the customer and switch between views to prove that the data binding is still functioning correctly.
13. Return to Visual Studio and stop debugging.

The ViewModel provides access to customer information through the `Current` property, but currently, it doesn't supply a way to navigate between customers. You can implement methods that increment and decrement the `currentCustomer` variable so that the `Current` property retrieves different customers, but you should do so in a manner that does not tie the view to the ViewModel.

The most commonly accepted technique is to use the Command pattern. In this pattern, the ViewModel exposes methods in the form of commands that the view can invoke. The trick is to avoid explicitly referencing these methods by name in the code for the view. To do this, XAML makes it possible for you to declaratively bind commands to the actions triggered by controls in the UI, as you'll see in the exercises in the next section.

Adding commands to a ViewModel

The XAML markup that binds the action of a control to a command requires that commands exposed by a ViewModel implement the `ICommand` interface. This interface defines the following items:

- **CanExecute** This method returns a Boolean value indicating whether the command can run. Using this method, a ViewModel can enable or disable a command depending on the context. For example, a command that fetches the next customer from a list should be able to run only if there's a next customer to fetch; if there are no more customers, the command should be disabled.
- **Execute** This method runs when the command is invoked.
- **CanExecuteChanged** This event is triggered when the state of the ViewModel changes. Under these circumstances, commands that could previously run might now be disabled and vice versa. For example, if the UI invokes a command that fetches the next customer from a list, and if that customer is the last customer, then subsequent calls to `CanExecute` should return `false` . In these circumstances, the `CanExecuteChanged` event should fire to indicate that the command has been disabled.

In the next exercise, you'll create a generic class that implements the `ICommand` interface.

To implement the Command class

1. In Visual Studio, right-click the **Customers** project, point to **Add**, and then select **Class**.
2. In the Add New Item – Customers dialog, select the **Class** template. In the **Name** box, type **Command.cs**, and then select **Add**.
3. In the Code and Text Editor window displaying the `Command.cs` file, add the following `using` directive to the list at the top of the file:

```
using System.Windows.Input;
```

The `ICommand` interface is defined in this namespace.

4. Make the `Command` class public and specify that it implements the `ICommand` interface, as follows in bold:

```
public class Command : ICommand  
{  
}
```

5. Add the following private fields to the `Command` class:

```
public class Command : ICommand  
{  
    private Action methodToExecute = null;  
    private Func<bool> methodToDetectCanExecute = null;  
}
```

The `Action` and `Func` types are briefly described in Chapter 20, “Decoupling application logic and handling events.” The `Action` type is a delegate that you can use to reference a method that takes no parameters and doesn’t return a value. The `Func<T>` type is also a delegate that can reference a method that takes no parameters but returns a value of the type specified by the type parameter `T`.

In this class, you’ll use the `methodToExecute` field to reference the code that the `Command` object will run when it’s invoked by the view. The `methodToDetectCanExecute` field will be used to reference the method that detects whether the command can run. (It may be disabled for some reason, depending on the state of the app or the data.)

6. Add a constructor to the `Command` class. This constructor should take two parameters: an `Action` object and a `Func<T>` object. Assign these parameters to the `methodToExecute` and `methodToDetectCanExecute` fields, as shown here in bold:

```
public Command : ICommand
{
    ...
    public Command(Action methodToExecute, Func<bool> methodToDetectCanExecute)
    {
        this.methodToExecute = methodToExecute;
        this.methodToDetectCanExecute = methodToDetectCanExecute;
    }
}
```

The `ViewModel` will create an instance of this class for each command. The `ViewModel` will supply the method to run the command and the method to detect whether the command should be enabled when it calls the constructor.

7. Implement the `Execute` and `CanExecute` methods of the `Command` class by using the methods referenced by the `methodToExecute` and `methodToDetectCanExecute` fields, as follows:

```
public Command : ICommand
{
    ...
    public Command(Action methodToExecute,
        Func<bool> methodToDetectCanExecute)
    {
        ...
    }

    public void Execute(object parameter)
    {
        this.methodToExecute();
    }

    public bool CanExecute(object parameter)
    {
        if (this.methodToDetectCanExecute is null)
        {
            return true;
        }
    }
}
```



```

        else
        {
            return this.methodToDetectCanExecute();
        }
    }
}

```

Notice that if the ViewModel provides a null reference for the `methodToDetectCanExecute` parameter of the constructor, the default action is to assume that the command can run, and the `CanExecute` method returns `true`.

8. Add the public `CanExecuteChanged` event to the `Command` class:

```

public Command : ICommand
{
    ...
    public bool CanExecute(object parameter)
    {
        ...
    }

    public event EventHandler CanExecuteChanged;
}

```

When you bind a command to a control, the control automatically subscribes to this event. This event should be raised by the `Command` object if the state of the `ViewModel` is updated and the value returned by the `CanExecute` method changes. The simplest strategy is to use a timer to raise the `CanExecuteChanged` event once a second or so. The control can then invoke `CanExecute` to determine whether the command can still be executed and take steps to enable or disable itself depending on the result.

9. Add the `using` directive shown next to the list at the top of the file:

```
using Windows.UI.Xaml;
```

10. Add the following field shown in bold to the `Command` class before the constructor:

```

public class Command : ICommand
{
    ...
    private Func<bool> methodToDetectCanExecute = null;
    private DispatcherTimer canExecuteChangedEventTimer = null;

    public Command(Action methodToExecute, Func<bool> methodToDetectCanExecute)
    {
        ...
    }
}

```

The `DispatcherTimer` class, defined in the `Windows.UI.Xaml` namespace, implements a timer that can raise an event at specified intervals. You'll use the `canExecuteChangedEventTimer` field to trigger the `CanExecuteChanged` event at one-second intervals.

- 11.** Add the `canExecuteChangedEventTimer_Tick` method shown in bold in the following code to the end of the `Command` class:

```
public class Command : ICommand
{
    ...
    public event EventHandler CanExecuteChanged;

    void canExecuteChangedEventTimer_Tick(object sender, object e)
    {
        if (this.CanExecuteChanged is not null)
        {
            this.CanExecuteChanged(this, EventArgs.Empty);
        }
    }
}
```

This method simply raises the `CanExecuteChanged` event if at least one control is bound to the command. Strictly speaking, this method should also check whether the state of the object has changed before raising the event. However, you'll set the timer interval to a lengthy period (in processing terms) to minimize any inefficiencies in not checking for a change in state.

- 12.** In the `Command` constructor, add the following statements shown in bold:

```
public class Command : ICommand
{
    ...
    public Command(Action methodToExecute, Func<bool> methodToDetectCanExecute)
    {
        this.methodToExecute = methodToExecute;
        this.methodToDetectCanExecute = methodToDetectCanExecute;

        this.canExecuteChangedEventTimer = new DispatcherTimer();
        this.canExecuteChangedEventTimer.Tick +=
            canExecuteChangedEventTimer_Tick;
        this.canExecuteChangedEventTimer.Interval = new TimeSpan(0, 0, 1);
        this.canExecuteChangedEventTimer.Start();
    }
    ...
}
```

This code initiates the `DispatcherTimer` object and sets the interval for timer events to one second before it starts the timer running.

- 13.** On the **Build** menu, select **Build Solution** and ensure that your app builds without errors.

You can now use the `Command` class to add commands to the `ViewModel` class. In the next exercise, you'll define commands to enable a user to move between customers in the view.

To add NextCustomer and PreviousCustomer commands to the ViewModel class

1. In Visual Studio, open the ViewModel.cs file in the Code and Text Editor window.
2. Add the following using directive to the top of the file and modify the definition of the ViewModel class to implement the INotifyPropertyChanged interface:

```
...
using System.ComponentModel;

namespace Customers
{
    public class ViewModel : INotifyPropertyChanged
    {
        ...
    }
}
```

3. Add the PropertyChanged event and OnPropertyChanged method to the end of the ViewModel class. This is the same code that you included in the Customer class.

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged is not null)
        {
            PropertyChanged(this,
                new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Remember that the view references data through the Current property in the data-binding expressions for the various controls that it contains. When the ViewModel class moves to a different customer, it must raise the PropertyChanged event to notify the view that the data to be displayed has changed.

4. Add the following fields and properties to the ViewModel class immediately after the constructor:

```
public class ViewModel : INotifyPropertyChanged
{
    ...
    public ViewModel()
    {
        ...
    }

    private bool _isAtStart;
    public bool IsAtStart
    {
        get => this._isAtStart;
        set
        {

```

```

        this._isAtStart = value;
        this.OnPropertyChanged(nameof(IsAtStart));
    }
}

private bool _isAtEnd;
public bool IsAtEnd
{
    get => this._isAtEnd;
    set
    {
        this._isAtEnd = value;
        this.OnPropertyChanged(nameof(IsAtEnd));
    }
}
...
}

```

You'll use these two properties to track the state of the ViewModel. The `IsAtStart` property will be set to `true` when the `currentCustomer` field in the `ViewModel` is positioned at the start of the customers collection, and the `IsAtEnd` property will be set to `true` when the `ViewModel` is positioned at the end of the customers collection.

5. Modify the constructor to set the `IsAtStart` and `IsAtEnd` properties, as shown here in bold:

```

public ViewModel()
{
    this.currentCustomer = 0;
    this.IsAtStart = true;
    this.IsAtEnd = false;
    this.customers = new List<Customer>
    {
        ...
    };
}

```

6. After the `Current` property, add the `Next` and `Previous` private methods shown in bold to the `ViewModel` class:

```

public class ViewModel : INotifyPropertyChanged
{
    ...
    public Customer Current
    {
        ...
    }

    private void Next()
    {
        if (this.customers.Count - 1 > this.currentCustomer)
        {
            this.currentCustomer++;
            this.OnPropertyChanged(nameof(Current));
            this.IsAtStart = false;
            this.IsAtEnd = (this.customers.Count - 1 == this.currentCustomer);
        }
    }
}

```

```

private void Previous()
{
    if (this.currentCustomer > 0)
    {
        this.currentCustomer--;
        this.OnPropertyChanged(nameof(Current));
        this.IsAtEnd = false;
        this.IsAtStart = (this.currentCustomer == 0);
    }
}
...
}

```



Note The Count property returns the number of items in a collection, but remember that the items in a collection are numbered from 0 to Count - 1.

These methods update the currentCustomer variable to refer to the next (or previous) customer in the customers list. Notice that these methods maintain the values for the IsAtStart and IsAtEnd properties and indicate that the current customer has changed by raising the PropertyChanged event for the Current property. These methods are private because they should not be accessible from outside the ViewModel class. External classes will run these methods by using commands, which you'll add in the following steps.

7. Add the NextCustomer and PreviousCustomer automatic properties to the ViewModel class, as shown here in bold:

```

public class ViewModel : INotifyPropertyChanged
{
    private List<Customer> customers;
    private int currentCustomer;
    public Command NextCustomer { get; private set; }
    public Command PreviousCustomer { get; private set; }
    ...
}

```

The view will bind to these Command objects so that the user can navigate between customers.

8. In the ViewModel constructor, set the NextCustomer and PreviousCustomer properties to refer to new Command objects, as follows:

```

public ViewModel()
{
    this.currentCustomer = 0;
    this.IsAtStart = true;
    this.IsAtEnd = false;
    this.NextCustomer = new Command(this.Next, () =>
        this.customers.Count > 1 && !this.IsAtEnd);
    this.PreviousCustomer = new Command(this.Previous, () =>
        this.customers.Count > 0 && !this.IsAtStart);
    this.customers = new List<Customer>
    {
        ...
    };
}

```

The `NextCustomer` Command object specifies the `Next` method as the operation to perform when the `Execute` method is invoked. The lambda expression `() => { return this.customers.Count > 1 && !this.IsAtEnd; }` is specified as the function to call when the `CanExecute` method runs.

This expression returns `true` as long as the customers list contains more than one customer and the `ViewModel` isn't positioned on the final customer in this list. The `PreviousCustomer` Command object follows the same pattern: it invokes the `Previous` method to retrieve the previous customer from the list. The `CanExecute` method references the expression `() => { return this.customers.Count > 0 && !this.IsAtStart; }`, which returns `true` as long as the customers list contains at least one customer and the `ViewModel` isn't positioned on the first customer in this list.

9. On the **Build** menu, select **Build Solution** and verify that your app still builds without errors.

Now that you've added the `NextCustomer` and `PreviousCustomer` commands to the `ViewModel`, you can bind these commands to buttons in the view. When the user selects a button, the appropriate command will run.

Microsoft publishes guidelines for adding buttons to views in UWP apps. The general recommendation is that buttons that invoke commands should be placed on a command bar. UWP apps provide two command bars: one at the top of the form and the other at the bottom. Buttons that navigate through an app or data are commonly placed on the top command bar; this is the approach that you'll adopt in the next exercise.



Note You can find the Microsoft guidelines for implementing command bars at <https://docs.microsoft.com/windows/apps/design/controls/command-bar>.

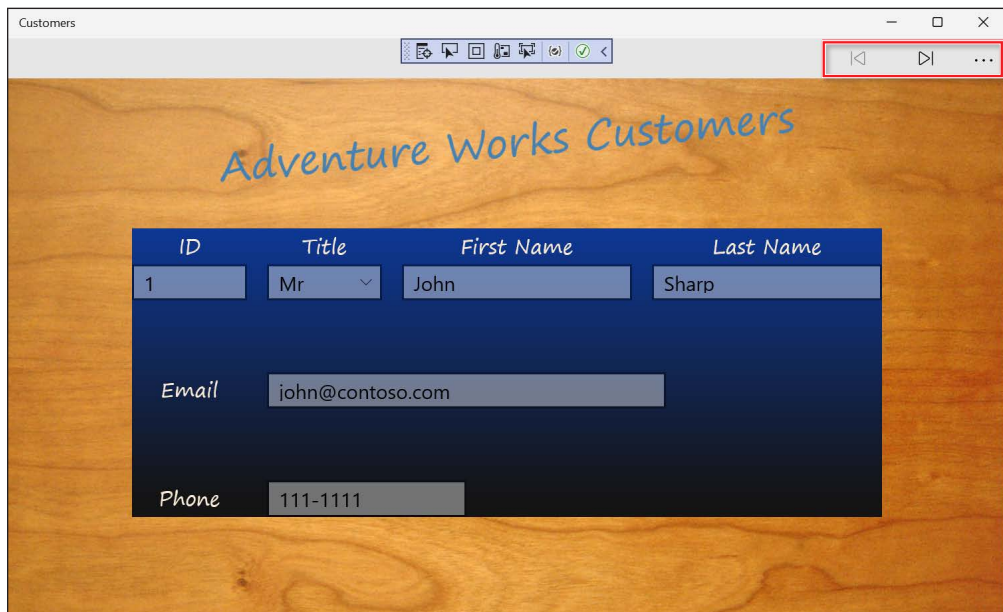
To add Next and Previous buttons to the Customers form

1. Open the `MainPage.xaml` file in the Design View window.
2. Scroll to the bottom of the **XAML** pane and add the following markup shown in bold, immediately after the final `</Grid>` tag but before the closing `</Page>` tag:

```
...
</Grid>
<Page.TopAppBar>
  <CommandBar>
    <AppBarButton x:Name="previousCustomer" Icon="Previous"
Label="Previous" Command="{Binding Path=PreviousCustomer}"/>
    <AppBarButton x:Name="nextCustomer" Icon="Next"
Label="Next" Command="{Binding Path=NextCustomer}"/>
  </CommandBar>
</Page.TopAppBar>
</Page>
```

There are several points to notice in this fragment of XAML markup:

- By default, the command bar appears at the top of the screen and displays icons for the buttons it contains. The label for each button is displayed only when the user selects the More (...) button that appears on the right side of the command bar. However, if you're designing an application that could be used across multiple countries or cultures, you should not provide hard-coded values for labels. Instead, store the text for these labels in a culture-specific resources file and bind the `Label` property dynamically when the application runs.
 - The `CommandBar` control can contain only a limited set of controls (controls that implement the `ICommandBarElement` interface). This includes the `AppBarButton`, `AppBarToggleButton`, and `AppBarSeparator` controls. These controls are specifically designed to operate within a `CommandBar` control. If you attempt to add a control such as a button to a command bar, you'll receive the error message `The specified value cannot be assigned to the collection.`
 - The UWP app templates include a variety of stock icons (such as for Previous and Next, shown in the preceding sample code) that you can display on an `AppBarButton` control. You can also define your own icons and bitmaps.
 - Each button has a `Command` property, which is the property that you can bind to an object that implements the `ICommand` interface. In this application, you have bound the buttons to the `PreviousCustomer` and `NextCustomer` commands in the `ViewModel1` class. When the user selects either of these buttons at runtime, the corresponding command will run.
3. On the **Debug** menu, select **Start Debugging**. The Customers form should appear and display the details for John Sharp. The command bar should be displayed at the top of the form and contain the Next and Previous buttons, highlighted in the upper right of the following image:



Notice that the Previous button is not available. This is because the `IsAtStart` property of the `ViewModel` is `true`, and the `CanExecute` method of the `Command` object referenced by the Previous button indicates that the command cannot run.

4. Select the ellipsis button on the command bar. The labels for the buttons should appear. These labels will be displayed until you select one of the buttons on the command bar.
5. On the command bar, select **Next**.

The details for customer 2, Diana Sharp, should appear, and after a short delay (of up to one second), the Previous button should become available. The `IsAtStart` property is no longer `true`, so the `CanExecute` method of the command returns `true`. However, the button is not notified of this change in state until the timer object in the command expires and triggers the `CanExecuteChanged` event, which might take up to a second to occur.



Note If you require a more instantaneous reaction to the change in state of commands, you can arrange for the timer in the `Command` class to expire more frequently. However, avoid reducing the time by too much; raising the `CanExecuteChanged` event too frequently can affect the performance of the UI.

6. On the command bar, select **Next** again.
7. The details for customer 3, Francesca Sharp, should appear, and after a short delay of up to one second, the Next button should no longer be available. This time, the `IsAtEnd` property of the `ViewModel` is `true`, so the `CanExecute` method of the `Command` object for the Next button returns `false`, and the command is disabled.
8. Resize the window to display the app in the narrow view and verify that the app continues to function correctly. The Next and Previous buttons should step forward and backward through the list of customers.
9. Return to Visual Studio and stop debugging.

Summary

In this chapter, you learned how to display data on a form by using data binding. You saw how to set the data context for a form and how to create a data source that supports data binding by implementing the `INotifyPropertyChanged` interface. You also learned how to use the Model-View-ViewModel pattern to create a UWP app, and you saw how to create a `ViewModel` with which a view can interact with a data source by using commands.

Quick reference

To	Do this
Bind the property of a control to the property of an object	Use a data-binding expression in the XAML markup of the control. For example: <code><TextBox ... Text="{Binding FirstName}" .../></code>
Enable an object to notify a binding of a change in a data value	Implement the <code>INotifyPropertyChanged</code> interface in the class that defines the object and raise the <code>PropertyChanged</code> event each time a property value changes. For example: <pre>class Customer : INotifyPropertyChanged { ... public event PropertyChangedEventHandler PropertyChanged; protected virtual void OnPropertyChanged(string propertyName) { if (PropertyChanged is not null) { PropertyChanged(this, new PropertyChangedEventArgs(propertyName)); } } }</pre>
Enable a control that uses data binding to update the value of the property to which it is bound	Configure the data binding as two-way. For example: <code><TextBox ... Text="{Binding FirstName, Mode=TwoWay}" .../></code>
Separate the business logic that runs when a user selects a Button control from the user interface that contains the Button control	Use a ViewModel that provides commands implemented with the <code>ICommand</code> interface and bind the Button control to one of these commands. For example: <code><Button x:Name="nextCustomer" ... Command="{Binding Path=NextCustomer}" /></code>

Accessing a remote database from a Universal Windows Platform app

After completing this chapter, you will be able to:

- Retrieve data from a database by creating a Representational State Transfer (REST) web service that provides remote access to a database through an entity model.
- Connect to a REST web service from a UWP app.
- Fetch data from a remote database by using a REST web service.
- Insert, update, and delete data in a remote database by using a REST web service.

Chapter 26, “Displaying and searching for data in a Universal Windows Platform app,” showed how to implement the Model-View-ViewModel (MVVM) pattern. It also explained how to separate the business logic of an app from the user interface (UI) by using a `ViewModel1` class that provides access to the data in the model and implements commands that the UI can use to invoke the logic of the app. Chapter 26 also illustrated how to use data binding to display the data presented by the `ViewModel` and how the UI can update this data. This all results in a fully functional Universal Windows Platform (UWP) app. In this chapter, you’ll turn your attention to the *model* aspect of the MVVM pattern. In particular, you’ll see how to implement a model that a UWP app can use to retrieve and update data in a remote database.

Retrieving data from a database

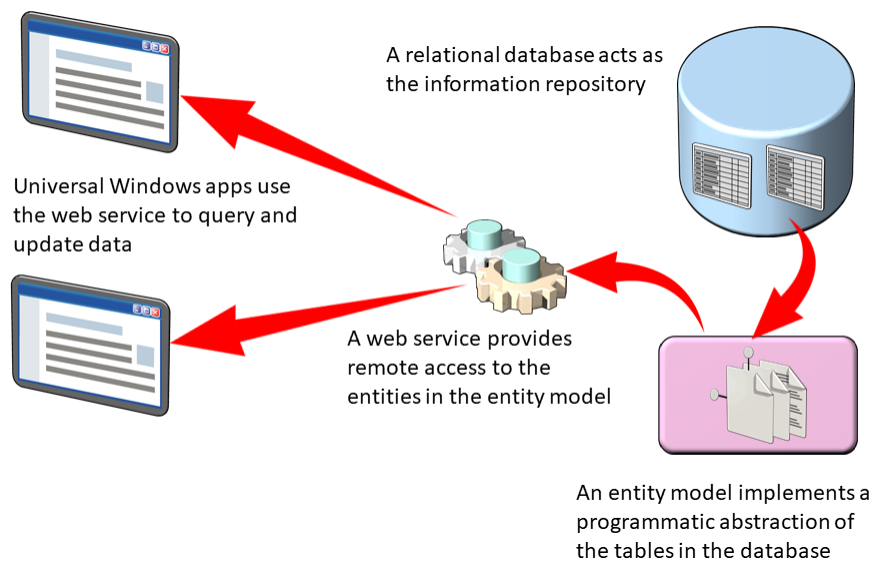
So far, the data you’ve used has been confined to a simple collection embedded in the `ViewModel` of the app. In the real world, the data displayed and maintained by an app is more likely to be stored in a data source such as a relational database.

UWP apps cannot directly access a relational database by using technologies provided by Microsoft (although some third-party database solutions are available). This might sound like a severe restriction, but there are sensible reasons for this limitation. Primarily, it eliminates dependencies that a UWP app might have on external resources, making the app a standalone item that can be easily packaged and downloaded from the Windows Store without requiring users to install and configure a database-management system on their computer. Additionally, many Windows 11 devices are resource constrained and don’t have the memory or disk space available to run a local database-management

system. However, many business apps will still require access to a database. To address this scenario, you can use a *web service*.

Web services can implement a variety of functions, but one common scenario is to provide an interface with which an app can connect to a remote data source to retrieve and update data. A web service can be located almost anywhere—from the computer the app is running on to a web server hosted on a computer on a different continent. As long as you can connect to the web service, you can use it to provide access to the repository of your information.

Microsoft Visual Studio provides templates and tools with which you can quickly and easily build a web service. The simplest strategy is to base the web service on an entity model generated by using the Entity Framework, as shown in the following diagram. The Entity Framework is a powerful technology with which you can connect to a relational database. It can reduce the amount of code needed to add data access capabilities to an app.



Note There isn't sufficient space in this book to go into great detail on how to use the Entity Framework, and the exercises in this section walk you through only the most essential steps to get started. If you want more information, see the article "Entity Framework 6" on the Microsoft website at <https://docs.microsoft.com/ef/ef6/>.

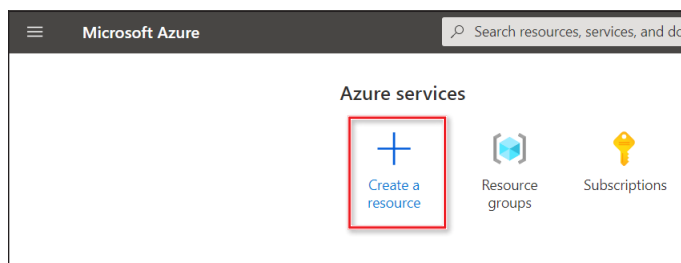
In this chapter, you'll continue with the same UWP app you worked on in the previous two chapters, called Customers, for a fictitious company called Adventure Works. To begin, you'll set up an AdventureWorks database, which contains the details of Adventure Works customers. To make the scenario more realistic, the exercises in this chapter show you how to create the database in the cloud by using Microsoft Azure SQL Database and how to deploy the web service to Azure. This architecture is common to many commercial apps, including e-commerce applications, mobile-banking services, and even video-streaming systems.



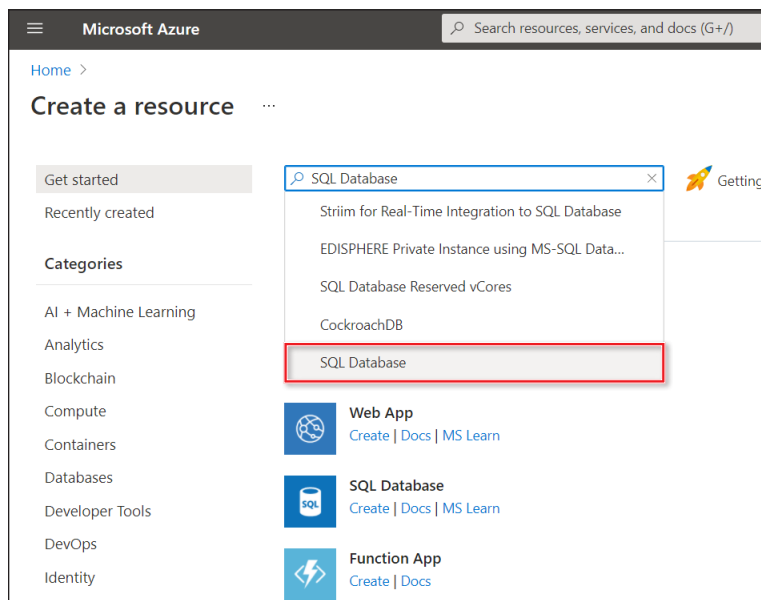
Note The exercises in this chapter require that you have an Azure account and subscription. If you don't already have an Azure account, you can sign up for a free trial account at <https://azure.microsoft.com/pricing/free-trial/>. Additionally, Azure requires that you have a valid Microsoft account it can associate your Azure account with. You can sign up for a Microsoft account at <https://signup.live.com/>.

To create an Azure SQL Database server and install the AdventureWorks sample database

1. Using a web browser, connect to the Azure portal at <https://portal.azure.com>, and sign in using your Microsoft account.
2. In the toolbar at the top of the portal, select **Create a resource**.



3. On the Create a resource page, in the **Search services and marketplace** box, type **SQL Database**, and select **SQL Database** in the dropdown list that appears.



4. On the SQL Database page, select **Create**.

5. On the **Basics** tab of the Create SQL Database page, perform the following tasks:

- Leave the **Subscription** box set to the name of your Azure subscription.
- Under the **Resource group** box, select **Create new**, and type **awgroup**.
- In the **Database name** box, type **AdventureWorks**.
- Under the **Server** box, select **Create new**. Then, in the **New Server** pane, type a unique name for your server. Use your company name or even your own name; I used **csharpstep-bystep2022**. (If the name you enter has been used by someone else, you will be alerted, in which case enter another name.) Enter a name and password for the administrator login (make a note of these items; I used **JohnSharp**, but I'm not going to tell you my password), select the location closest to you, and then select **OK** to return to the SQL Database pane.
- For **Want to use SQL elastic pool**, select **No**.
- For the pricing tier, select **Configure database**. On the Configure page, in the **Service tier** dropdown list box, select **Basic** (this is the cheapest option), and then select **Apply**.



Important Do not select any pricing tier other than Basic, and do not enable SQL elastic pool unless you want to receive a potentially significant bill at the end of the month. For information about SQL Database pricing, see <https://azure.microsoft.com/pricing/details/sql-database/>.

- Under **Backup storage redundancy**, select **Geo-redundant backup storage** (this is the default option).

6. At the bottom of the page, select **Next: Networking**.

The screenshot shows the 'Create SQL Database' page in the Microsoft Azure portal. The breadcrumb trail is 'Home > Create a resource > SQL Database >'. The page title is 'Create SQL Database' with a Microsoft logo and a link to 'Learn more'. Below the title, it says 'provision with smart defaults, or visit each tab to customize. Learn more'. The page is divided into several sections: 'Project details', 'Database details', and 'Backup storage redundancy'. In the 'Project details' section, 'Subscription' and 'Resource group' are selected. In the 'Database details' section, 'Database name' is 'AdventureWorks' and 'Server' is '(new) csharpstepbystep2022 (UK South)'. Under 'Want to use SQL elastic pool?', 'No' is selected. Under 'Compute + storage', 'Basic' is selected. In the 'Backup storage redundancy' section, 'Geo-redundant backup storage' is selected. A warning message states: 'Selected value for backup storage redundancy is Geo-redundant backup storage. Note that database backups will be geo-replicated to the paired region. Learn more'. An information message states: 'Your use of either of the Preview backup storage redundancy options (ZRS and LRS) is governed by the agreement under which you obtained Microsoft Azure Services. By selecting a Preview redundancy option, you confirm that you agree to the preview terms in such agreement. Microsoft Azure Legal Information: Learn more'. At the bottom, there are two buttons: 'Review + create' and 'Next: Networking >'. The 'Next: Networking >' button is highlighted with a red rectangle.

Microsoft Azure Search resources, services, and docs (G+/I)

Home > Create a resource > SQL Database >

Create SQL Database

Microsoft

provision with smart defaults, or visit each tab to customize. [Learn more](#)

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * (New) [Redacted] [Create new](#)

Resource group * (New) awgroup [Create new](#)

Database details

Enter required settings for this database, including picking a logical server and configuring the compute and storage resources

Database name * AdventureWorks ✓

Server * (new) csharpstepbystep2022 (UK South) [Create new](#)

Want to use SQL elastic pool? * ☐ Yes ☒ No

Compute + storage * **Basic**
2 GB storage [Configure database](#)

Backup storage redundancy

Choose how your PITR and LTR backups are replicated. Geo restore or ability to recover from regional outage is only available when geo-redundant storage is selected.

Backup storage redundancy * ☐ Locally-redundant backup storage - Preview
☐ Zone-redundant backup storage - Preview
☒ Geo-redundant backup storage

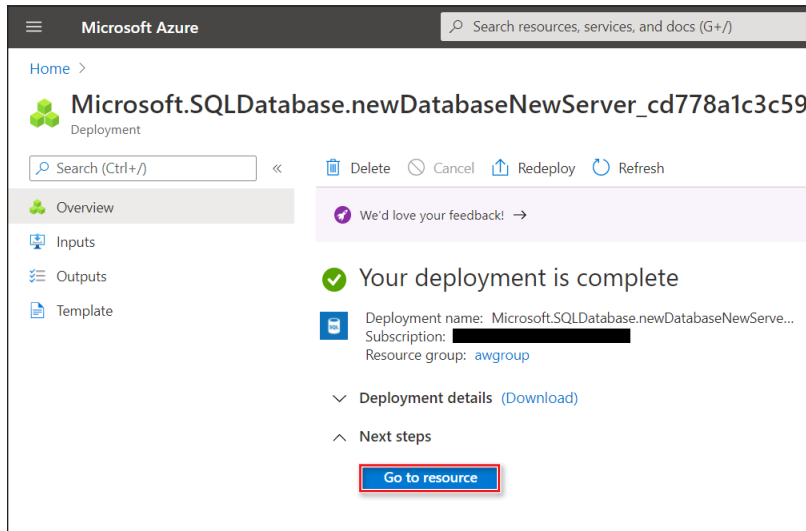
Warning: Selected value for backup storage redundancy is Geo-redundant backup storage. Note that database backups will be geo-replicated to the paired region. [Learn more](#)

Info: Your use of either of the Preview backup storage redundancy options (ZRS and LRS) is governed by the agreement under which you obtained Microsoft Azure Services. By selecting a Preview redundancy option, you confirm that you agree to the preview terms in such agreement. Microsoft Azure Legal Information: [Learn more](#)

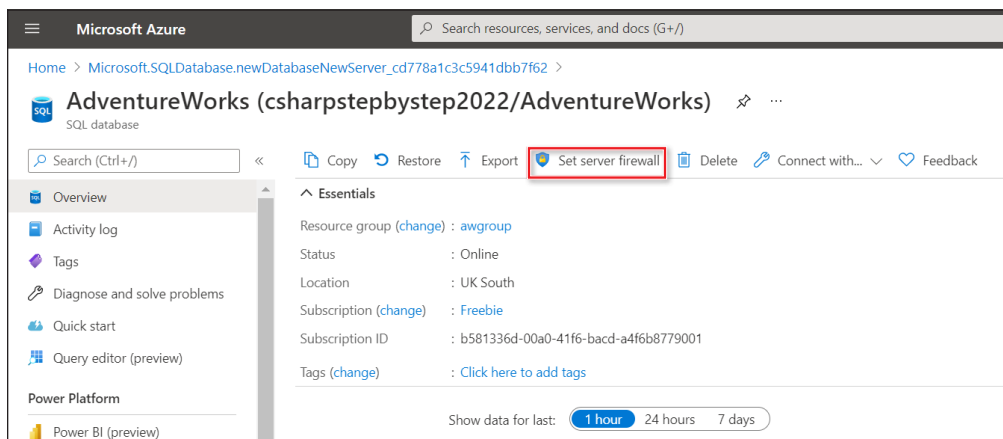
[Review + create](#) [Next: Networking >](#)

7. On the Networking tab of the Create SQL Database page, accept the default settings and select **Next: Security**.
8. On the Security tab of the Create SQL Database page, accept the default settings and select **Next: Additional settings**.

9. On the Additional settings tab of the Create SQL Database page, under **Use existing data**, select **Sample**. This will create the AdventureWorksLT database, containing sample customers and other data. Then select **Review + create**.
10. On the final Create SQL Database page, select **Create**. Wait while the database server is created and the database is deployed.
11. When deployment is complete, select **Go to resource**.



12. On the AdventureWorks database page, in the toolbar at the top of the page, select **Set server firewall**.



13. On the Firewall settings page, select **Add client IP**, and select **Allow Azure services and resources to access this server**. These settings enable you to access the database from your computer, as well as allowing any web services that you create to connect to the database.

Microsoft Azure

Home > Microsoft.SQLDatabase.newDatabaseNewServer_cd778a1c3c5941dbb7f62 > AdventureWorks (cs

Firewall settings

sharpstepbystep2022 (SQL server)

Save Discard **+ Add client IP**

Deny public network access ⓘ
☐ Yes ☒ No

Click here to create a new private endpoint.
[Create Private Endpoint](#)

Minimum TLS Version ⓘ
☐ 1.0 ☐ 1.1 ☒ 1.2

Connection Policy ⓘ
☒ Default ☐ Proxy ☐ Redirect

Allow Azure services and resources to access this server ⓘ
☒ Yes ☐ No

Client IP address 90.248.64.88

Rule name	Start IP	End IP
		...
ClientIPAddress_2021-7-...		...

Virtual networks
[+ Add existing virtual network](#) [+ Create new virtual network](#)

Rule name	Virtual network	Subnet	Address Range
No vnet rules for this server.			

14. Select **Save**. Verify that the message “Successfully updated server firewall rules” appears, and then select **OK**.



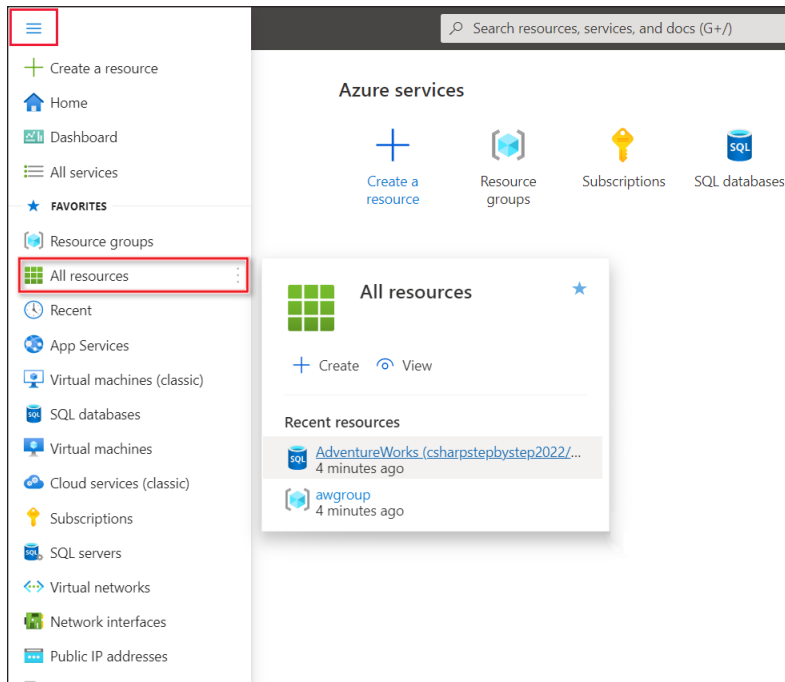
Note These steps are important. Without them, you won’t be able to connect to the database from applications running on your computer. You can also create firewalls that span a range of IP addresses if you need to open access to a set of computers.

The sample AdventureWorks database contains a table named Customer in the SalesLT schema. This table includes the columns containing the data presented by the Customers UWP app and several others. Using the Entity Framework, you can choose to ignore columns that are not relevant, but you won’t be able to create new customers if any of the columns you ignore do not allow nulls and do not have default values. In the Customer table, this restriction applies to the NameStyle, PasswordHash, and PasswordSalt columns (used to encrypt user passwords) as well as the rowguid and ModifiedDate

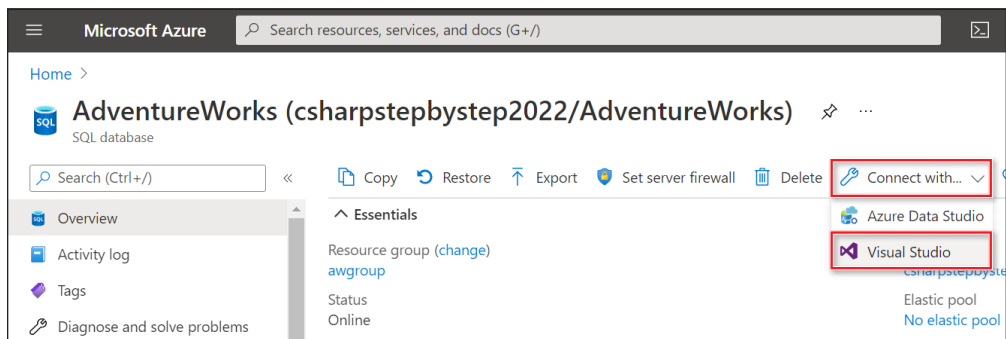
columns (used by other sample apps from Microsoft that reference this table). To avoid complications and to enable you to focus on the functionality of the app itself, in the next exercise, you'll remove these columns from the Customer table.

To remove unneeded columns from the AdventureWorks database

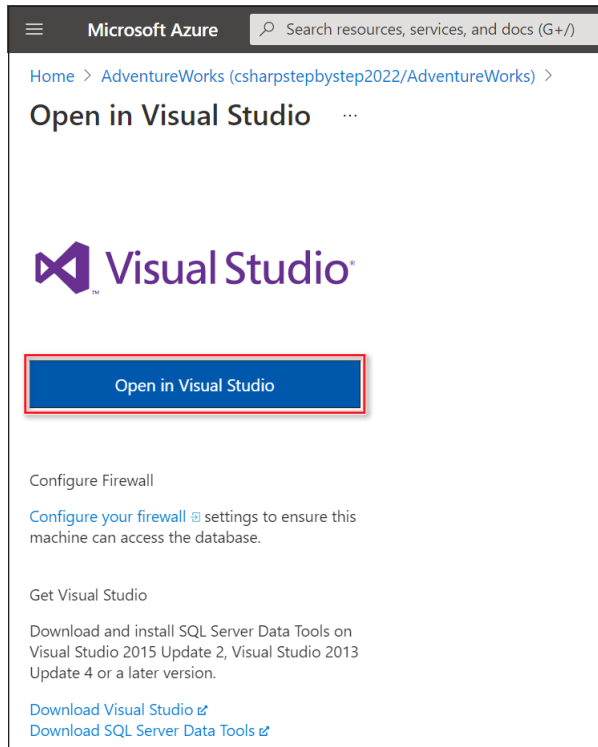
1. In the Azure portal, select the navigation menu in the top-left corner, select **All resources**, and then select the AdventureWorks database.



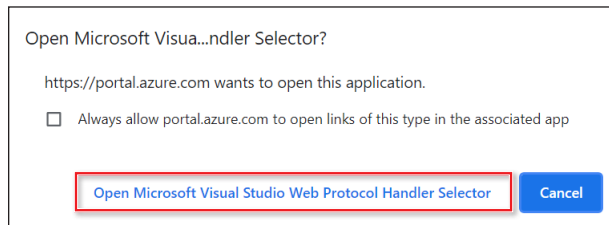
2. In the toolbar above the AdventureWorks SQL Database pane, select **Connect with**, and then select **Visual Studio**.



3. On the Open in Visual Studio page, select **Open in Visual Studio**.

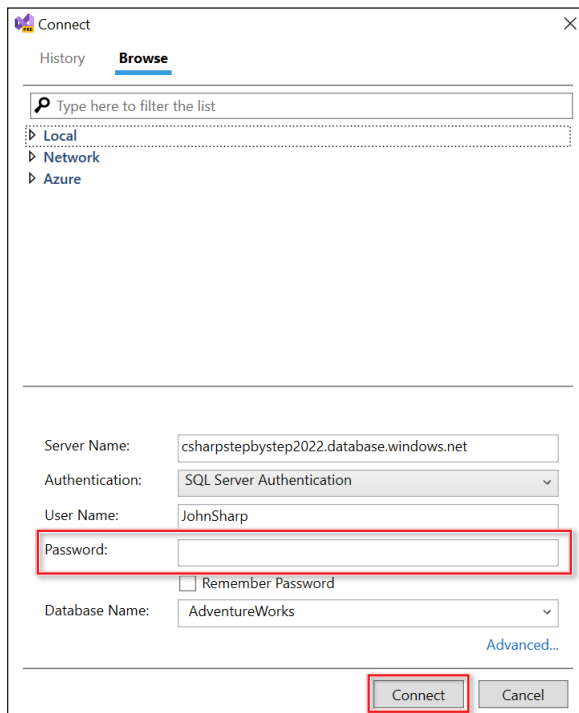


4. If prompted, select **Open Microsoft Visual Studio Web Protocol Handler Selector**.

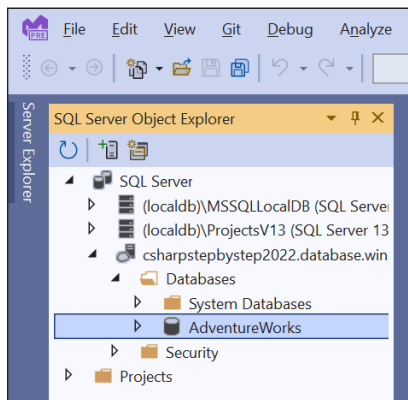


Visual Studio will start and prompt you to connect to the database.

5. In the Connect dialog, enter the administrator password that you specified earlier, and then select **Connect**.



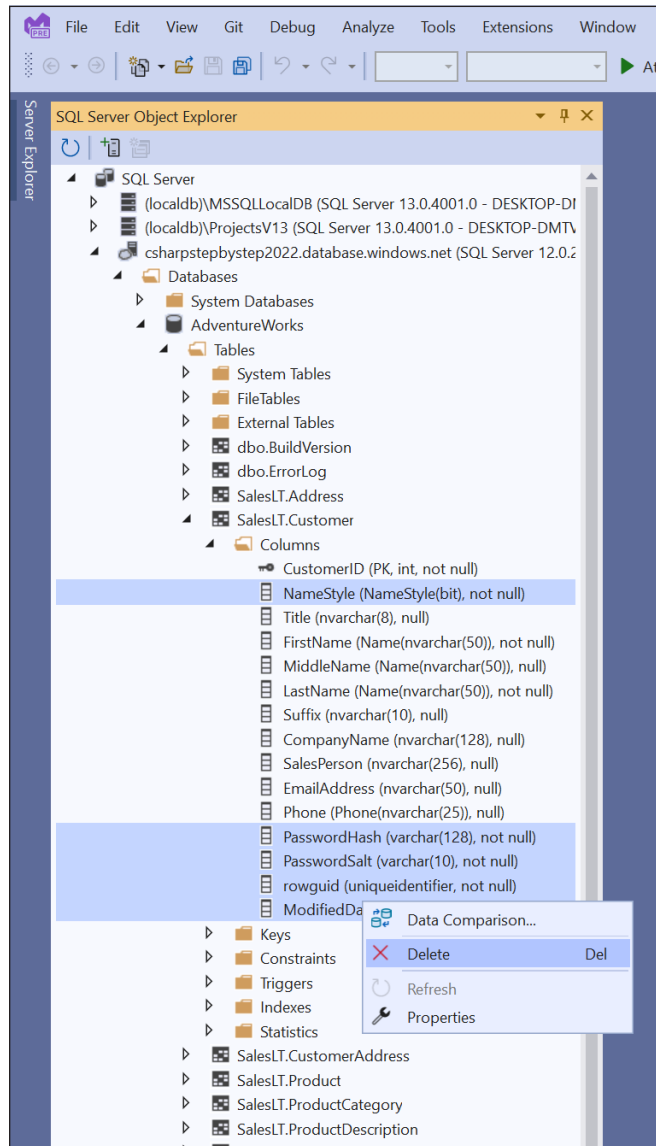
Visual Studio connects to the database, which appears in the SQL Server Object Explorer window on the left side of the Visual Studio IDE.



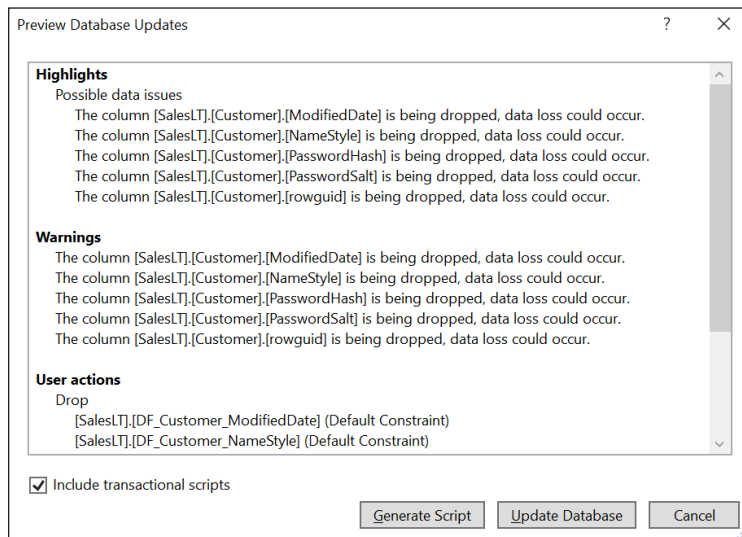
6. In the SQL Server Object Explorer pane, expand the **AdventureWorks** database, expand **Tables**, expand **SalesLT.Customer**, and then expand **Columns**.

The columns in the table are listed. The columns that aren't used by the application and that disallow null values must be removed.

7. Select the **NameStyle** column, press the **Ctrl** key, and then select the **PasswordHash**, **PasswordSalt**, **rowguid**, and **ModifiedDate** columns.
8. Right-click any of the selected columns, and then select **Delete**.



Visual Studio analyzes these columns. In the Preview Database Updates dialog, it displays a list of warnings and other issues that could occur if the columns are removed.



9. In the Preview Database Updates dialog, select **Update Database**.
10. When the columns have been removed, close the SQL Server Object Explorer pane, but leave Visual Studio 2022 open.

Creating an entity model

Now that you've created the AdventureWorks database in the cloud, you can use the Entity Framework to create an entity model that an app can use to query and update information in this database. If you have worked with databases in the past, you might be familiar with technologies such as ADO.NET, which provides a library of classes that you can use to connect to a database and run SQL commands. ADO.NET is useful but requires a decent understanding of SQL, and if you're not careful, it can force you into structuring your code around the logic necessary to perform SQL commands instead of focusing on the business operations of your app. The Entity Framework provides a level of abstraction that reduces the dependencies that your apps have on SQL.

Essentially, the Entity Framework implements a mapping layer between a relational database and your app. It generates an entity model that consists of collections of objects that your app can use just as it would any other collection. A collection typically corresponds to a table in the database, and each row in a table corresponds to an item in the collection. You perform queries by iterating through the items in a collection, usually with Language-Integrated Query (LINQ). Behind the scenes, the entity model converts your queries into SQL SELECT commands that fetch the data. You can modify the data in the collection, and then you can arrange for the entity model to generate and perform the appropriate SQL INSERT, UPDATE, and DELETE commands to perform the equivalent operations in the database. In short, the Entity Framework is an excellent vehicle for connecting to a database and retrieving and managing data without requiring you to embed SQL commands in your code.

In the following exercise, you'll create a web API project that you'll use to provide access to an entity model for the Customer table in the AdventureWorks database.

To create the AdventureWorks web API project

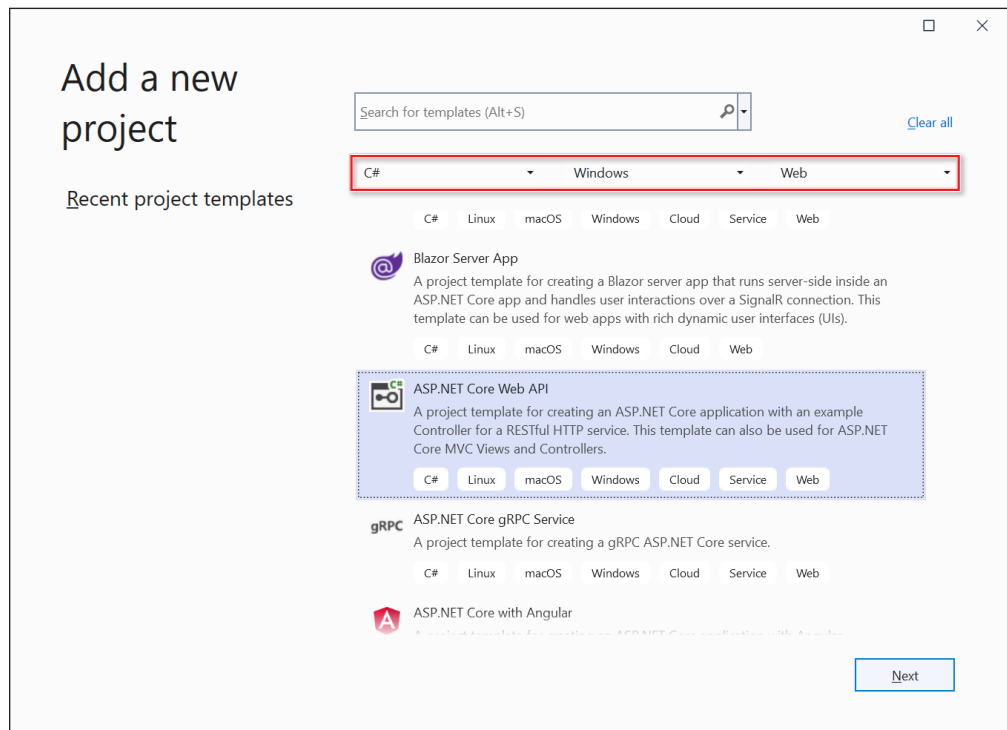
1. In Visual Studio, open the **Customers** solution, located in the **\Microsoft Press\VCSBS\Chapter 27\Web Service** folder in your **Documents** folder.

This project contains a modified version of the Customers app from Chapter 26. The ViewModel implements additional commands, which let a user navigate to the first or last customer in the customers collection, and the command bar contains First and Last buttons that invoke these commands. The sample data created by the constructor in the ViewModel has been removed.

2. In Solution Explorer, right-click the **Customers** solution (not the Customers project), point to **Add**, and then select **New Project**.
3. In the Add a New Project dialog, select **C#** in the **language** dropdown list box, select **Windows** in the **platform** dropdown list box, select **Web** in the **project type** dropdown list box, scroll down and select the **ASP.NET Core Web API** template, and then select **Next**.



Note Make sure you don't accidentally select the ASP.NET Core Web App template.



4. In the Configure your new project dialog, set the **Project name** to **AdventureWorksService**. Accept the default location, and then select **Next**.
5. In the Additional information dialog, select the **.NET 6.0** framework. Leave the remaining options at their default values, and then select **Create**.

Visual Studio creates the new project and adds it to the Customers solution.
6. The default Web API project contains sample code that you don't need for this application. In Solution Explorer, select the **WeatherForecast.cs** file and delete it. Then expand the **Controllers** folder, select the **WeatherForecastController.cs** file and delete it too.
7. On the **Build** menu, select **Rebuild Solution**. This process will ensure that any cached references in the project to the items you have just deleted are removed.

As mentioned, you cannot access a relational database directly from a UWP app. Instead, you have created a web API project (this is not a UWP app), which will host the entity model that you create in the app. The web API template enables you to build a web service that you can host on Azure to allow client applications to connect quickly and easily. This web service will provide remote access to the entity model for the Customers UWP app. Visual Studio provides additional wizards and tools with which you can quickly create the code for the web service.

To create the Customers entity model

1. In Solution Explorer, right-click the **AdventureWorksService** project, point to **Add**, and then select **New folder**. Name the new folder **Models**.
2. In Solution Explorer, right-click the **Models** folder, point to **Add**, and then select **Class**.
3. In the Add New Item – AdventureWorksService dialog, in the **Name** box, type **CustomerModel.cs**, and then select **Add**.
4. In the Code and Text Editor window displaying the CustomerModel.cs file, delete the existing using directives from the top of the file and replace them with those shown here:

```
using System.ComponentModel.DataAnnotations;  
using System.ComponentModel.DataAnnotations.Schema;
```

5. Annotate the CustomerModel class with the Table attribute shown here:

```
[Table("Customer", Schema = "SalesLT")]  
public class CustomerModel  
{  
}
```

You use attributes to attach metadata to classes and other objects in C#. The `Table` attribute is used by the Entity Framework to associate the class with the specified table and schema in a database—in this case, the `Customer` table in the `SalesLT` schema. The Entity Framework uses this information to generate the appropriate SQL `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements when the application is built and run.



Note For a detailed description of the attributes used by the Entity Framework, read “Code First Data Annotations” at <https://docs.microsoft.com/ef/ef6/modeling/code-first/data-annotations>.

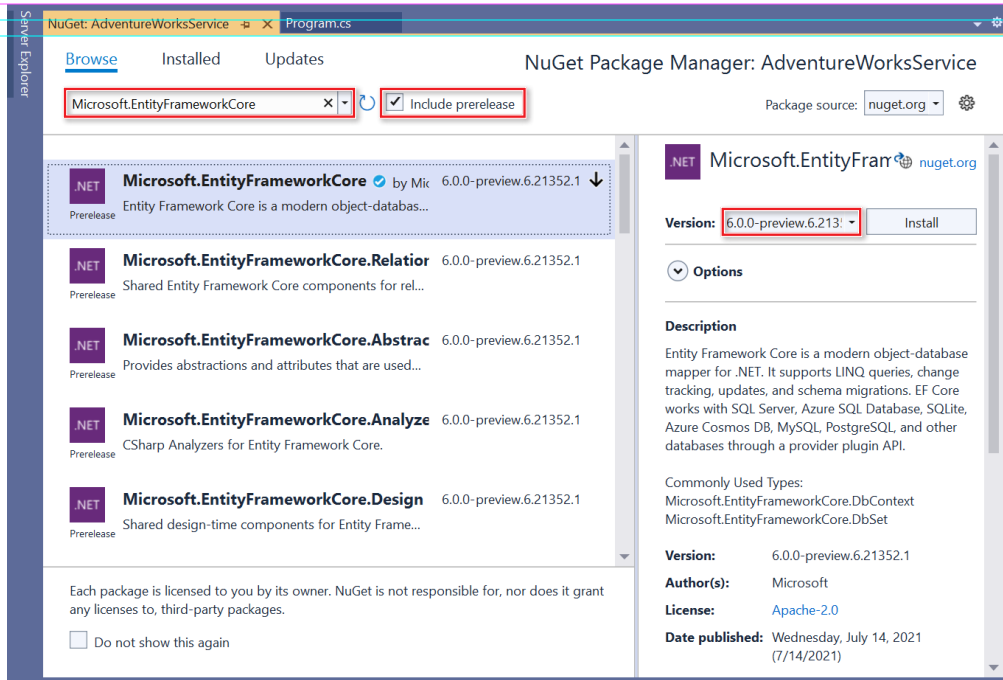
6. Add the following automatic properties shown in bold to the `CustomerModel` class. The `Key` attribute specifies that the `CustomerID` column is the primary key for the table in the database.

```
[Table("Customer", Schema = "SalesLT")]
public class CustomerModel
{
    [Key]
    public int CustomerID { get; set; }
    public string Title { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string EmailAddress { get; set; }
    public string Phone { get; set; }
}
```

7. In Solution Explorer, right-click the **AdventureWorksService** project, and then select **Manage NuGet Packages**.
8. In the NuGet Package Manager window, on the **Browse** tab, in the **Search** box, type **Microsoft.EntityFrameworkCore**. Select the **Microsoft.EntityFrameworkCore** package, and then select **Install** to add the package to the project.



Important Make sure you install version 6.0.0 or later. At the time of writing, the .NET 6 version of Entity Framework was only available as a pre-released package. If necessary, select the Include prerelease checkbox, and install the latest prerelease build of the package.



9. In the Preview Changes dialog, select **OK** to confirm the package installation. In the License Acceptance dialog, select **I Accept**.
10. Return to the **Browse** tab in the NuGet Package Manager, select the **Microsoft.EntityFrameworkCore.SqlServer** package, and then select **Install** to add the package to the project.
11. In the Preview Changes dialog, select **OK** to confirm the package installation. In the License Acceptance dialog, select **I Accept**.
12. In Solution Explorer, right-click the **Models** folder, point to **Add**, and then select **Class**. Name the class **CustomerContext.cs**.
13. In the Code and Text Editor window, replace the using directives at the top of the CustomerContext.cs file with the directive shown here:
using Microsoft.EntityFrameworkCore;
14. Modify the definition of the CustomerContext class to inherit from the DbContext class.

```
public class CustomerContext : DbContext
{
}
```

The DbContext class is part of the Entity Framework. This class provides the necessary plumbing and other bits of code to map from a C# class to a table in a database.

15. Add the constructor and Customers automatic property shown in the following code to the CustomerContext class:

```
public class CustomerContext : DbContext
{
    public CustomerContext(DbContextOptions<CustomerContext> options)
        : base(options)
    {
    }

    public DbSet<CustomerModel> Customers { get; set; }
}
```

The constructor simply calls the constructor in the base DbContext class to set up the required infrastructure when a new instance of the CustomerContext class is created. The Customers property provides the mapping between a set of Customers objects in your application code and the rows in the Customer table in the database.



Note For more information, see “Defining DbSets” at <https://docs.microsoft.com/ef/ef6/modeling/code-first/dbsets>.

16. In Solution Explorer, select the **Startup.cs** file for the AdventureWorks project to open it in the Code and Text Editor window.
17. Add the following using directive to the list at the top of the file:
- ```
using Microsoft.EntityFrameworkCore;
```
18. Find the ConfigureServices method. This method runs when the web service starts running. You use this method to set up the various items that the web service needs to run, including creating the DbContext object used to connect to the database.
19. Add the statement shown in the following code in bold to the ConfigureServices method. This statement creates a DbContext to connect to the AdventureWorks database. The connection details will be stored in a configuration file and can be accessed using the key AdventureWorksDB.

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddDbContext<CustomerContext>(optionsBuilder =>
 optionsBuilder.UseSqlServer(Configuration.GetConnectionString("AdventureWorksDB")));

 services.AddControllers();
 services.AddSwaggerGen(c =>
 {
 c.SwaggerDoc("v1", new OpenApiInfo { Title = "AdventureWorksService", Version = "v1"
 });
 });
}
```



**Note** The Swagger generator set up by the `ConfigureServices` method deploys a useful test harness that you can use to verify that the operations you implement in the web service are functioning correctly. You will use this feature later.

20. In Solution Explorer, select the **appsettings.json** file to open it in the Code and Text Editor window.
21. Add the connection string shown in the following code in bold to the file. This string contains the parameters needed to connect to the AdventureWorks database. In this string, replace the server name `csharpstepbystep2022` with the name of the server you set up when you created the database in Azure. Replace `JohnSharp` with your username, and `YourPassword` with your password.

```
{
 "ConnectionStrings": {
 "AdventureWorksDB": "Server=tcp:csharpstepbystep2022.database.windows.net,1433;Initial
Catalog=AdventureWorks;Persist Security Info=False;User ID=JohnSharp;Password=YourPassword;
MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection
Timeout=30;"
 },
 "Logging": {
 "LogLevel": {
 "Default": "Information",
 "Microsoft": "Warning",
 "Microsoft.Hosting.Lifetime": "Information"
 }
 },
 "AllowedHosts": "*"
}
```



**Note** In the real world, you should *never* save a password in this way. Instead, you should protect the password by storing it in an encrypted format.

22. On the **Build** menu, select **Rebuild Solution**. The solution should build without any errors.

## Creating and using a REST web service

You have created an entity model that you can use to retrieve and maintain customer information. The next step is to implement a web service so that a UWP app can access the entity model.

With Visual Studio 2022, you can create a web service in an ASP.NET Core web API project based on an entity model generated by the Entity Framework. The web service uses the entity model to retrieve data from a database and update the database.

A web service uses one or more controllers to expose operations to the internet. A controller is associated with an address in the web service, and you access resources through this address. The ASP.NET Core web API project creates web service controllers that implement the REST model. REST uses a navigational scheme to represent business objects and services over a network, and the HTTP protocol to transmit requests to access these objects and services.

A client app that accesses a resource submits a request in the form of a URL, which the web service parses and processes. For example, Adventure Works might publish customer information, exposing the details of each customer as a single resource, by using a scheme similar to this:

*<http://Adventure-Works.com/DataService/Customers/1>*

Accessing this URL causes the web service to retrieve the data for customer 1. This data can be returned in a number of formats, but for portability, the most common formats include XML and JavaScript Object Notation (JSON). A typical JSON response generated by a REST web service request issuing the previous query looks like this:

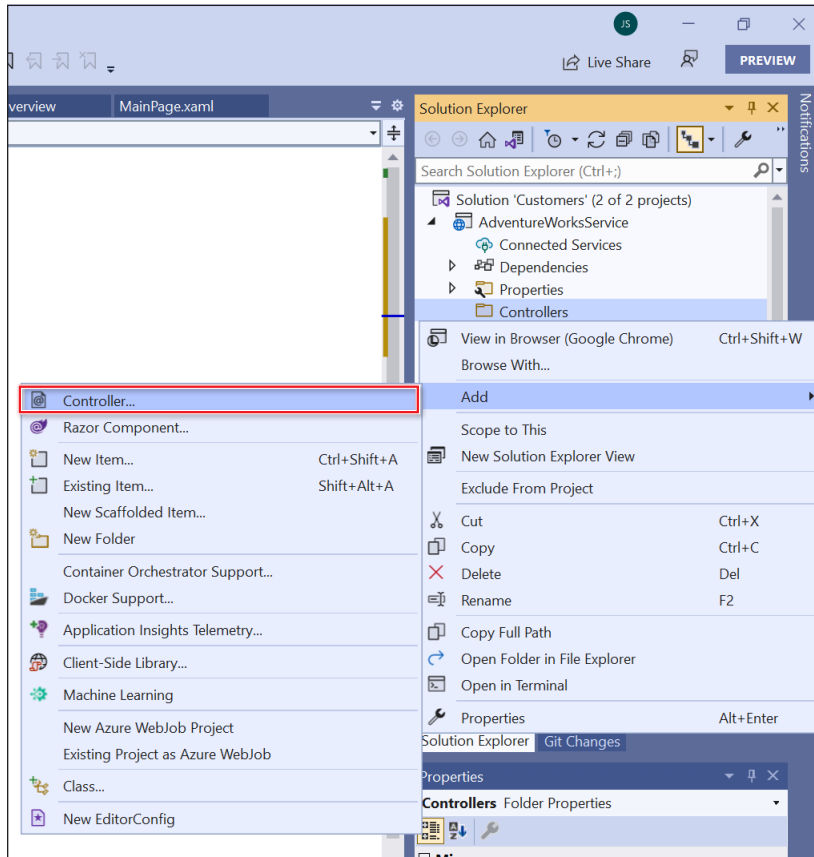
```
{
 "CustomerID": 1,
 "Title": "Mr",
 "FirstName": "Orlando",
 "LastName": "Gee",
 "EmailAddress": "orlando0@adventure-works.com",
 "Phone": "245-555-0173"
}
```

The REST model relies on the app that accesses the data to send the appropriate HTTP verb as part of the request to access the data. For example, the simple request shown previously should send an HTTP GET request to the web service. HTTP supports other verbs as well, such as POST, PUT, and DELETE, which you can use to create, modify, and remove resources, respectively. Writing the code to generate the appropriate HTTP requests and parsing the responses returned by a REST web service all sounds quite complicated. Fortunately, the ASP.NET Core web API provides libraries that can handle much of this complexity for you.

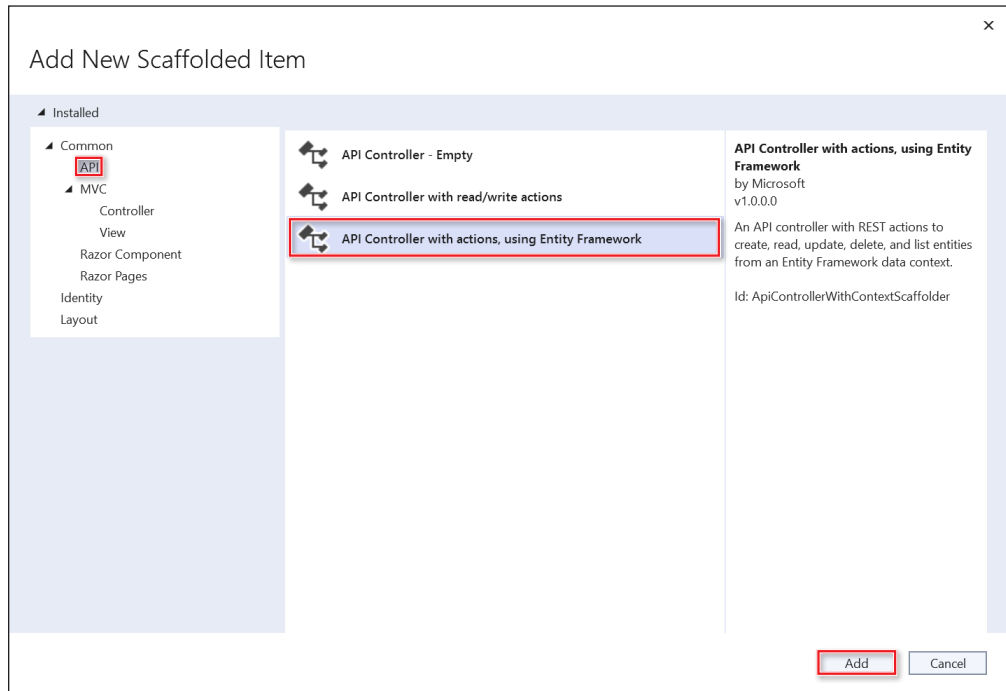
In the following exercise, you'll implement the REST web service operations for the AdventureWorks entity model. This web service will make it possible for a client app to query and maintain customer information.

## To create the AdventureWorks REST web service operations

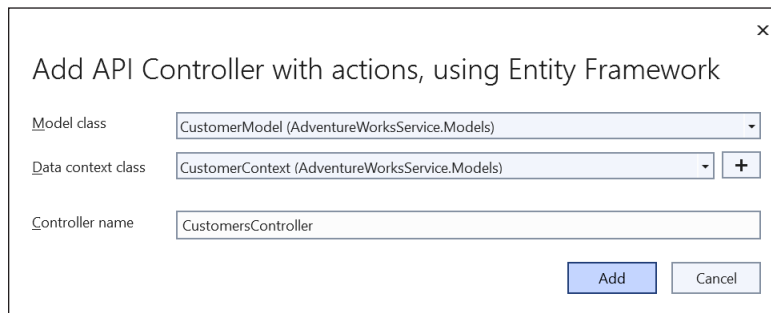
1. In Visual Studio, in the AdventureWorksService project, right-click the **Controllers** folder, point to **Add**, and then select **Controller...**



2. In the Add New Scaffolded Item dialog, in the left pane, select **API**. In the middle pane, select the **API Controller with actions, using Entity Framework** template. Then select **Add**.



3. In the Add API Controller dialog, in the **Model class** dropdown list, select **CustomerModel (AdventureWorksService.Models)**. In the **Data context class** dropdown list, select **CustomerContext (AdventureWorksService.Models)**. Change the **Controller name** to **CustomersController**, and then select **Add**.



In a web service created by using the ASP.NET web API template, all incoming web requests are handled by one or more controller classes, and each controller class exposes methods that map to the different types of REST requests for each of the resources that the controller exposes. The `CustomersController` class generated by the Scaffold wizard looks like this:

```
[Route("api/[controller]")]
[ApiController]
public class CustomersController : ControllerBase
{
 private readonly CustomerContext _context;

 public CustomersController(CustomerContext context)
 {
 _context = context;
 }

 // GET: api/Customers
 [HttpGet]
 public async Task<ActionResult<IEnumerable<CustomerModel>>> GetCustomers()
 {
 return await _context.Customers.ToListAsync();
 }

 // GET: api/Customers/5
 [HttpGet("{id}")]
 public async Task<ActionResult<CustomerModel>> GetCustomer(int id)
 {
 var customerModel = await _context.Customers.FindAsync(id);
 if (customerModel == null)
 {
 return NotFound();
 }

 return customerModel;
 }

 // PUT: api/Customers/5
 // To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
 [HttpPut("{id}")]
 public async Task<IActionResult> PutCustomer(int id, CustomerModel customerModel)
 {
 if (id != customerModel.CustomerID)
 {
 return BadRequest();
 }

 _context.Entry(customerModel).State = EntityState.Modified;

 try
 {
 await _context.SaveChangesAsync();
 }
 catch (DbUpdateConcurrencyException)
 {
 }
 }
}
```

```

 if (!CustomerExists(id))
 {
 return NotFound();
 }
 else
 {
 throw;
 }
 }

 return NoContent();
}

// POST: api/Customers
// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
[HttpPost]
public async Task<ActionResult<CustomerModel>> PostCustomer(
 CustomerModel customerModel)
{
 _context.Customers.Add(customerModel);
 await _context.SaveChangesAsync();

 return CreatedAtAction("GetCustomerModel",
 new { id = customerModel.CustomerID }, customerModel);
}

// DELETE: api/Customers/5
[HttpDelete("{id}")]
public async Task<IAActionResult> DeleteCustomer(int id)
{
 var customerModel = await _context.Customers.FindAsync(id);
 if (customerModel == null)
 {
 return NotFound();
 }

 _context.Customers.Remove(customerModel);
 await _context.SaveChangesAsync();

 return NoContent();
}

private bool CustomerExists(int id)
{
 return _context.Customers.Any(e => e.CustomerID == id);
}
}

```

The `GetCustomers` method handles requests to retrieve all customers. It satisfies these requests by simply returning the entire `Customers` collection (`DbSet`) from the Entity Framework data model that you created previously. Behind the scenes, the Entity Framework fetches all the customers from the database and uses this information to populate the `Customers` collection. This method is invoked if an app sends an HTTP GET request to the `api/Customers` URL in this web service.



The `GetCustomer` method (not to be confused with `GetCustomers`) takes an integer parameter. This parameter specifies the `CustomerID` of a specific customer, and the method uses the Entity Framework to find the details of this customer before returning it. `GetCustomer` runs when an app sends an HTTP GET request to the `api/Customers/n` URL, where *n* is the ID of the customer to retrieve.

The `PutCustomer` method runs when an app sends an HTTP PUT request to the web service. The request specifies a customer ID and the details of a customer, and the code in this method uses the Entity Framework to update the specified customer with the details.

The `PostCustomer` method responds to HTTP POST requests and takes the details of a customer as its parameter. This method adds a new customer with these details to the database (the details are not shown in the preceding code sample).

Finally, the `DeleteCustomer` method handles HTTP DELETE requests and removes the customer with the specified customer ID.

Notice that all these methods are asynchronous. Connecting to a web service over the internet and fetching data from a database may take a few seconds, and you don't want to tie up a thread in the web service while this processing takes place.



**Note** The code generated by the web API template optimistically assumes that it will always be able to connect to the database. In the world of distributed systems, however, where the database and web service are located on separate servers, this might not always be the case. Networks are prone to transient errors and timeouts; a connection attempt might fail because of a temporary glitch and succeed if it's retried a short time later. Reporting a temporary glitch to a client as an error can be frustrating to the user. If possible, it might be better to silently retry the failing operation as long as the number of retries is not excessive. (You don't want the web service to freeze if the database is really unavailable.) For detailed information on this strategy, see "Cloud Service Fundamentals Data Access Layer: Transient Fault Handling" at <http://social.technet.microsoft.com/wiki/contents/articles/18665.cloud-service-fundamentals-data-access-layer-transient-fault-handling.aspx>.

The ASP.NET web API template automatically generates code that directs requests to the appropriate method in the controller classes. You can add more controller classes if you need to manage other resources, such as products or orders.

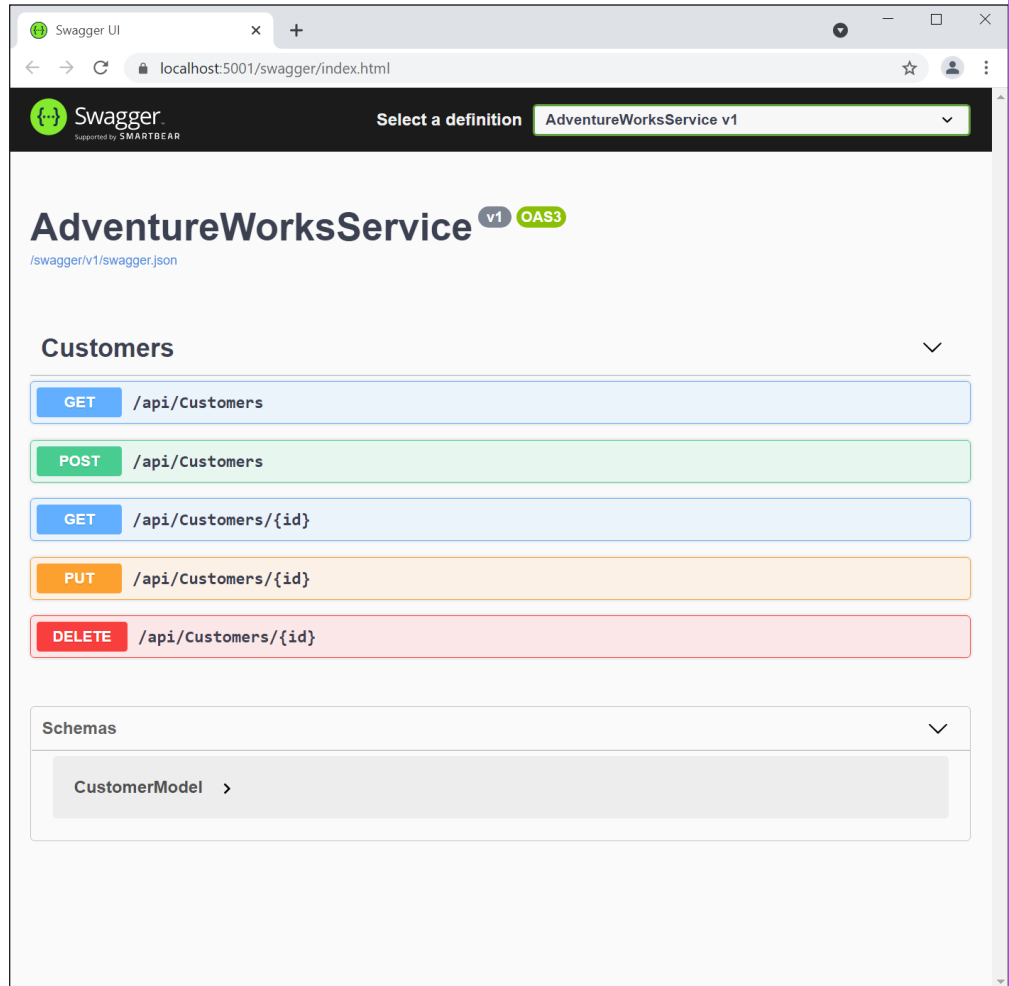
You can also create controller classes manually by using the same pattern as that shown by the `CustomersController` class—you don't have to fetch and store data in a database by using the Entity Framework. The ASP.NET Core Web API template contains a sample controller in the `ValuesController.cs` file, which you can copy and augment with your own code.

4. In Solution Explorer, right-click the **AdventureWorksService** project, point to **Debug**, and select **Start New Instance**.

This action starts the IISExpress web server, which hosts the web service.

5. If the Trust ASP.NET Core SSL Certificate message box appears, select **Yes**, and allow Visual Studio to install a certificate. IISExpress uses this certificate to support local testing of SSL connections.

The Debug Console window opens, displaying diagnostic information. Your web browser should also open and display a page that looks like the following image. This is the Swagger test page. It displays the operations that the web service provides and enables you to test them.



6. Select the **GET** button for the `/api/Customers` operation. (Remember that this operation retrieves the details for every customer.) Select **Try it out** and then select **Execute** to view the results. The operation should return a response body that contains a JSON array holding the customer information. You can scroll through the results.

The screenshot shows the Swagger UI interface in a web browser. The URL is `localhost:5001/swagger/index.html`. The interface displays the `GET /api/Customers` endpoint. The **Parameters** section shows "No parameters". The **Execute** button has been clicked, and the **Responses** section shows a 200 status code. The response body is a JSON array of customer objects, which is displayed in a dark-themed code editor. The response body contains four customer objects, each with fields: `customerID`, `title`, `firstName`, `lastName`, `emailAddress`, and `phone`.

```
[
 {
 "customerID": 1,
 "title": "Mr.",
 "firstName": "Orlando",
 "lastName": "Gee",
 "emailAddress": "orlando@adventure-works.com",
 "phone": "245-555-0173"
 },
 {
 "customerID": 2,
 "title": "Mr.",
 "firstName": "Keith",
 "lastName": "Harris",
 "emailAddress": "keith@adventure-works.com",
 "phone": "170-555-0127"
 },
 {
 "customerID": 3,
 "title": "Ms.",
 "firstName": "Donna",
 "lastName": "Carreras",
 "emailAddress": "donna@adventure-works.com",
 "phone": "279-555-0130"
 },
 {
 "customerID": 4,
 "title": "Mr.",
 "firstName": "James",
 "lastName": "Cox",
 "emailAddress": "james@adventure-works.com",
 "phone": "455-555-0141"
 }
]
```

- On the Swagger page, scroll down to the **GET** button for the `/api/Customers/{id}` operation. This operation returns the details of a single customer. Select **GET**, and then select **Try it out**. In the **Parameters** section, specify an **id** of **55**, and then select **Execute**. The details for Eric Lang should be displayed.

The screenshot shows the Swagger UI in a web browser at `localhost:5001/swagger/index.html`. The interface is for the `GET /api/Customers/{id}` endpoint. In the **Parameters** section, the `id` parameter is set to `55`. The **Execute** button has been clicked, and the **Responses** section is expanded. It shows the **curl** command used, the **Request URL**, and the **Server response**. The response is a 200 status code with a JSON body containing customer details for Eric Lang. The response headers and a table of responses are also visible.

**Parameters**

| Name                 | Description |
|----------------------|-------------|
| <b>id</b> * required |             |
| integer(int32)       |             |
| (path)               |             |

**Execute** **Clear**

**Responses**

**curl**

```
curl -X GET "https://localhost:5001/api/Customers/55" -H "accept: text/plain"
```

**Request URL**

```
https://localhost:5001/api/Customers/55
```

**Server response**

| Code | Details                                                                                                                                                                                                                                                                                                                                 |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 200  | <p><b>Response body</b></p> <pre>{  "customerID": 55,  "title": "Mr.",  "firstName": "Eric",  "lastName": "Lang",  "emailAddress": "eric6@adventure-works.com",  "phone": "932-555-0163"}</pre> <p><b>Response headers</b></p> <pre>content-type: application/json; charset=utf-8date: Wed28 Jul 2021 17:09:10 GMTserver: Kestrel</pre> |

**Responses**

| Code | Description | Links    |
|------|-------------|----------|
| 200  | Success     | No links |

8. The POST, PUT, and DELETE buttons enable you to test insert, update, and delete operations against the web service. Feel free to experiment with them.
9. When you've finished, close the web browser, close the Debug Console window, and return to Visual Studio.

As it stands, although the web API is perfectly functional, it's not the most scalable service. For example, if your database contains thousands of customers, do you really want the GET operation to retrieve them all? This could take time and consume considerable resources. In this situation, most modern applications will retrieve data in chunks of maybe 100 rows at a time. The user can still fetch and browse every row if they want, but if they're really only interested in the first few dozen rows, then retrieving every customer is a waste.

Additionally, the web API enables users to search for a customer by using the customer ID, but in many cases it would be more useful to be able to retrieve a customer by other attributes, such as by name or even by the email address. In the next exercises, you'll modify the web API to meet these requirements.

### To retrieve blocks of customers

1. In the CustomersController.cs file, find the GetCustomers method.
2. Change the signature of this method to take two parameters: an offset and a count. The method will retrieve count customers, starting at row offset in the database.

```
[HttpGet]
public async Task<ActionResult<IEnumerable<CustomerModel>>> GetCustomers(
 int offset, int count)
{
 ...
}
```

3. Modify the code in the body of the method as shown here in bold:

```
[HttpGet]
public async Task<ActionResult<IEnumerable<CustomerModel>>> GetCustomers(
 int offset, int count)
{
 if (offset < 0 || count < 0)
 {
 return BadRequest();
 }

 return await _context.Customers.
 OrderBy(r => r.CustomerID).
 Skip(offset).
 Take(count).ToListAsync();
}
```

This code quickly checks to make sure that the offset and count values aren't negative. If the parameters are valid, the method uses the Entity Framework to retrieve the data. The Skip method jumps over the number of rows specified, and the Take method fetches the required rows. The order in which rows are stored in a relational database such as SQL Server is not

necessarily guaranteed, so retrieving count rows starting at position offset could return a random collection of customers. To counter this problem, the `OrderBy` method ensures that the data is sorted by `CustomerID` first.

## To implement enhanced searching

1. Create another method named `FindCustomers` in the `CustomersController` class. This method should return a list of customers and take a set of parameters that define the search criteria. The method will return all customers that match these criteria. The search criteria have default values of `%`. This character is the SQL wildcard character. If the user omits any of the parameters, they will be treated as match any value.

```
// GET: api/Customes/find
[HttpGet("find")]
public async Task<ActionResult<IEnumerable<CustomerModel>>> FindCustomers(
 string title = "%", string firstName = "%", string lastName = "%",
 string email = "%", string phone = "%")
{
}
```

Notice that the function has the `HttpGet` attribute with the relative URI of "find". A user invokes this function by using the *api/Customers/find* address in the web service.

2. Add the following bold code to the body of the `FindCustomers` method:

```
public async Task<ActionResult<IEnumerable<CustomerModel>>> FindCustomers(
 string title = "%", string firstName = "%", string lastName = "%",
 string email = "%", string phone = "%")
{
 var query = from c in _context.Customers
 where EF.Functions.Like(c.Title, title) &&
 EF.Functions.Like(c.FirstName, firstName) &&
 EF.Functions.Like(c.LastName, lastName) &&
 EF.Functions.Like(c.EmailAddress, email) &&
 EF.Functions.Like(c.Phone, phone)
 select c;

 return await query.ToListAsync();
}
```

This code runs a LINQ query that uses the static `Like` function provided as part of the Entity Framework library to find all rows that match the criteria passed in as the parameters. The `Like` function is part of the static `EF` class in the Entity Framework library.

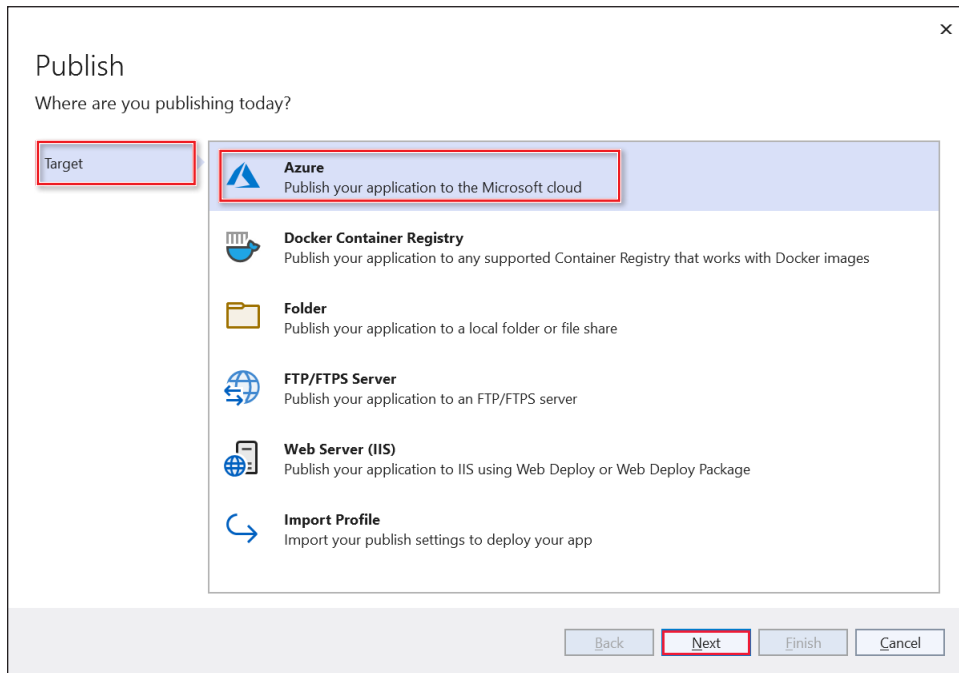
3. On the **Build** menu, select **Rebuild Solution**. Verify that the web service rebuilds without any errors or warnings.
4. In Solution Explorer, right-click the **AdventureWorksService** project, point to **Debug**, and then select **Start New Instance**.
5. When the Swagger page appears, select the **GET** button for the `/api/Customers` operation, and then select **Try it out**.

6. Set the `offset` parameter to 10 and the `count` parameter to 5. Then select **Execute**. The operation should return five customers.
7. Scroll down the Swagger page, select **GET** for the `/api/Customers/find` operation, and then select **Try it out**.
8. In the `firstName` field, type **O%**, in the `lastName` field, type **%a%**, leave the other fields at their default value (%), and select **Execute**. This search should return all customers whose first name starts with O and that have an *a* somewhere in their last name. The query should return three customers, with IDs 348, 452, and 29511.
9. Try some more searches of your own. When you're finished, close the web browser, close the Debug Console window, and return to Visual Studio.

The web API is currently running using IISExpress on your computer. To make it available to other users, you should deploy it to Azure. You can do this by using the Publish wizard available with Visual Studio 2022, as described in the following exercise.

### To deploy the web service to the cloud

1. In Solution Explorer, right-click the **AdventureWorksService** project, and then select **Publish**.
2. In the Publish wizard, on the Target page, select **Azure**, and then select **Next**.



3. On the Specific target page, select **Azure App Service (Windows)**. Then select **Next**.

**Publish**

Which Azure service would you like to use to host your application?

Target

**Specific target**

- Azure App Service (Windows)**  
Publish your application code to a managed infrastructure that is easy to scale
- Azure App Service (Linux)**  
Publish your application code to a managed infrastructure that is easy to scale
- Azure App Service Container**  
Publish your application as a Docker image to Azure Container Registry and run it on Azure App Service
- Azure Container Registry**  
Publish your application as a Docker image to Azure Container Registry
- Azure Virtual Machine**  
Manage your own infrastructure

**Back** **Next** **Finish** **Cancel**

4. On the App Service page, sign in to your Azure account if prompted. Then select your Azure subscription. Above the **App Service instances** box, select the plus icon (+) to create a new Azure App Service.

**Publish**

Select existing or create a new Azure App Service

Microsoft account

Subscription name

Target

Specific target

**App Service**

View

Resource group

API Management

Search

App Service instances

(No resources found)

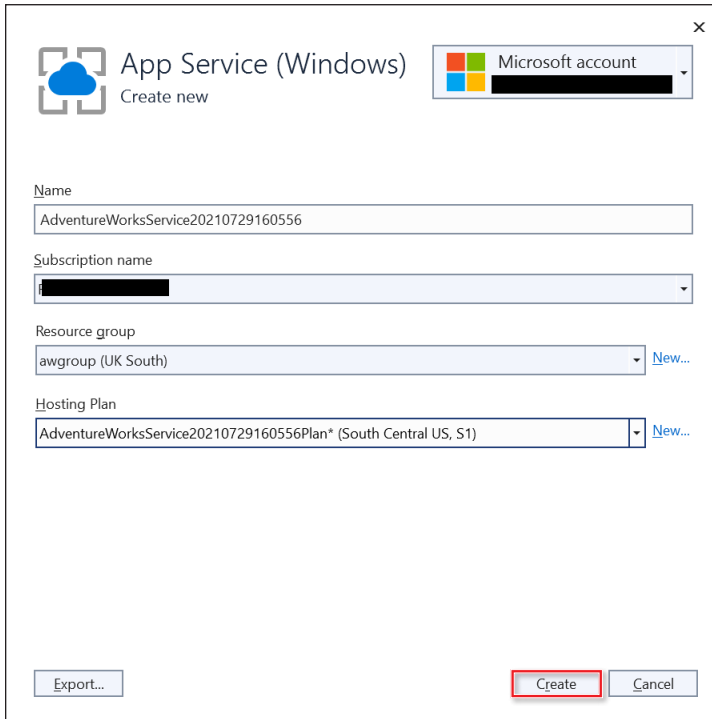
**+**

☐ Run from package file

**Back** **Next** **Finish** **Cancel**



5. In the App Service (Windows) dialog, accept the default app name, select your subscription, and set the resource group to **awgroup** (you created this resource group earlier, using the Azure portal). Then accept the default hosting plan name, and select **Create**.



The screenshot shows the 'App Service (Windows) Create new' dialog box. It has a title bar with a close button (X). The dialog contains the following fields and controls:

- Name:** A text input field containing 'AdventureWorksService20210729160556'.
- Subscription name:** A dropdown menu showing a partially obscured subscription name.
- Resource group:** A dropdown menu showing 'awgroup (UK South)' with a 'New...' link to the right.
- Hosting Plan:** A dropdown menu showing 'AdventureWorksService20210729160556Plan\* (South Central US, S1)' with a 'New...' link to the right.
- Buttons:** At the bottom, there are three buttons: 'Export...', 'Create' (highlighted with a red rectangle), and 'Cancel'.

The App Service provides the hosting environment for your web service. Behind the scenes, Microsoft hosts your API app on a web server.

6. Back in the Publish wizard, verify that the **App Service instances** box lists your new App Service instance, and then select **Next**.

**Publish**

Select existing or create a new Azure App Service

Microsoft account

Target: Subscription name

Specific target: View

App Service: Resource group

API Management: Search

AdventureWorksService20210729160556

App Service instances

- awgroup
  - AdventureWorksService20210729160556
    - Deployment Slots

☐ Run from package file

Back Next Finish Cancel

- On the API Management page, select **Skip this step**, and then select **Finish**.

**Publish**

Enable API consumption for teams, customers, and Logic and Power Apps

Microsoft account

Target: Subscription name

Specific target: View

App Service: Resource group

API Management: Search

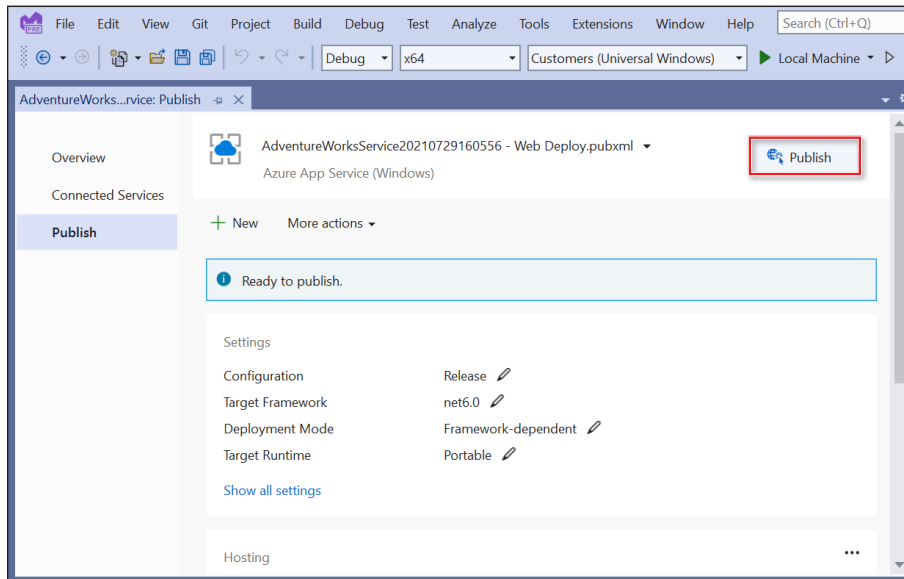
API Management APIs

(No resources found)

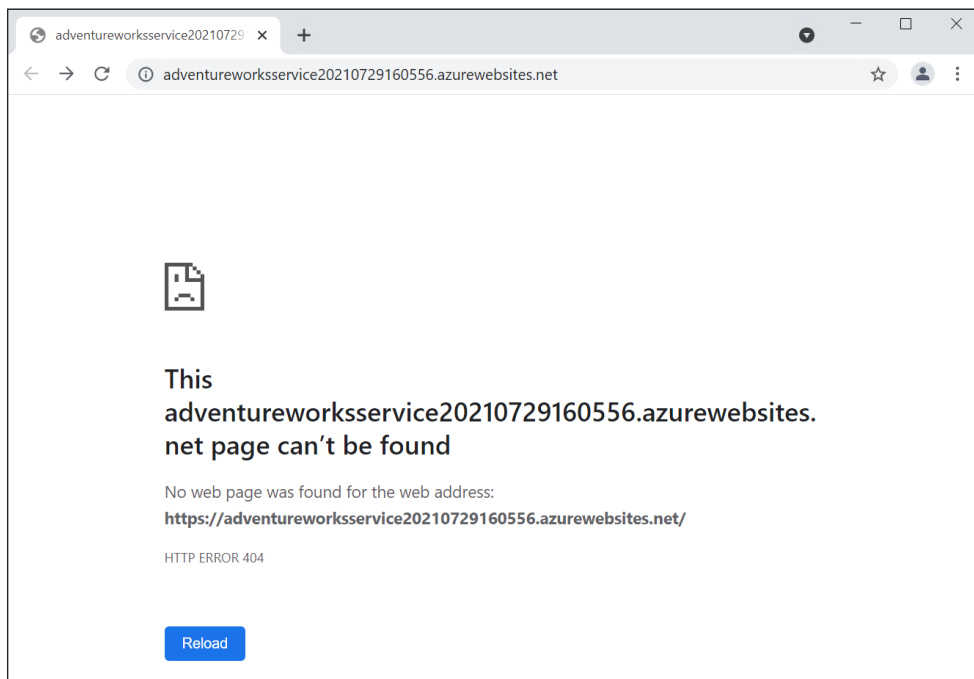
☒ Skip this step

Back Next Finish Cancel

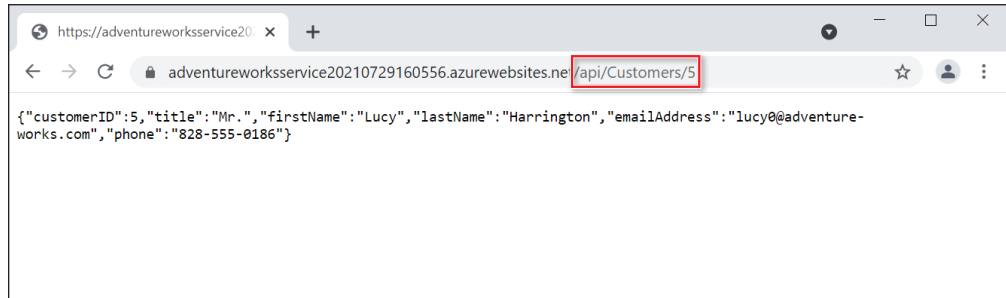
8. Wait for the wizard to finish. Then, in Visual Studio, on the Publish page, select **Publish**.



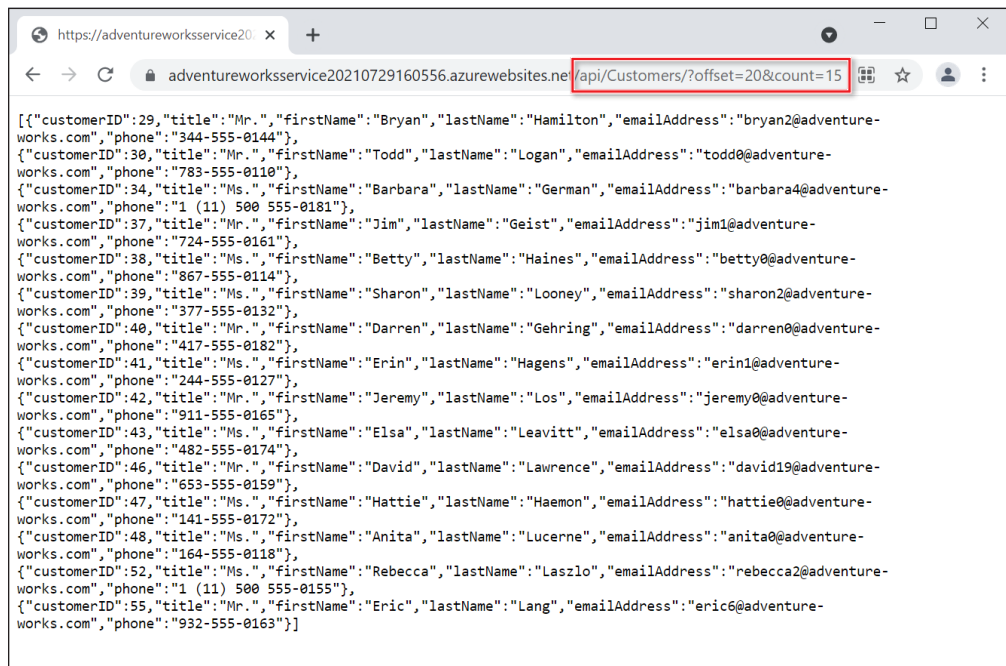
When the web service has been deployed, your web browser will start and most likely display an error page similar to that shown in the following image. This error occurs because there are no resources available through the default address of the web service; you access all data through the `api/Customers` URI.



9. In the address bar in the web browser, append the text `/api/Customers/5`, and then press **Enter**. This sends a GET request to the web service to retrieve the data for the customer with the customer ID of 5. The result is displayed as a JSON string:



10. In the address bar, change the URI to `/api/Customers/?offset=20&count=15` and press **Enter**. This URI sends a GET request to the method that takes the `offset` and `count` parameters. The results should be 15 rows of data, starting with the 20th customer found in the database (which actually has a customer ID of 29).



11. Close the web browser and return to Visual Studio.

## Updating the UWP application to use the web service

The next phase of this journey is to connect to the web service from the Customers UWP app and then use the web service to fetch some data. Before you do this, you'll update the UI in the Customers project. Sending a request to a web service could be a long-running operation, and you need to ensure that the user is aware that something is happening while this occurs. Additionally, sending a request over a network is an inherently error-prone operation; networks fail and time-out regularly, and other errors can occur. Therefore, you'll also add the ability for the UI to display error messages to the user.

### To enable the UI to support long-running operations and error reporting

1. In Solution Explorer, expand the **Customers** project, and open the **ViewModel.cs** file in the Code and Text Editor window.
2. Add the following fields and properties to the **ViewModel** class, after the constructor:

```
private bool _isBusy;
public bool IsBusy
{
 get => this._isBusy;
 set
 {
 this._isBusy = value;
 this.OnPropertyChanged(nameof(IsBusy));
 }
}

private string _lastError = null;
public string LastError
{
 get => this._lastError;
 private set
 {
 this._lastError = value;
 this.OnPropertyChanged(nameof(LastError));
 }
}
```

3. Add the properties shown next in bold to the start of the **ViewModel** class. (The fields not in bold should already exist.) The app will use these properties to indicate which record is currently being displayed by the UI.

```
private List<Customer> customers;
public int CustomerListCount { get => customers is null ? 0 : customers.Count; }
private int currentCustomer;
public int CurrentCustomerIndex { get => currentCustomer + 1; }
```

4. Update the code for the `Next`, `Previous`, `First`, and `Last` methods to indicate that the current record has changed when the user navigates between customers, as highlighted here in bold:

```
private void Next()
{
 if (this.customers.Count - 1 > this.currentCustomer)
 {
 this.currentCustomer++;
 this.OnPropertyChanged(nameof(Current));
 this.OnPropertyChanged(nameof(CurrentCustomerIndex));
 ...
 }
}

private void Previous()
{
 if (this.currentCustomer > 0)
 {
 this.currentCustomer--;
 this.OnPropertyChanged(nameof(Current));
 this.OnPropertyChanged(nameof(CurrentCustomerIndex));
 ...
 }
}

private void First()
{
 this.currentCustomer = 0;
 this.OnPropertyChanged(nameof(Current));
 this.OnPropertyChanged(nameof(CurrentCustomerIndex));
 ...
}

private void Last()
{
 this.currentCustomer = this.customers.Count - 1;
 this.OnPropertyChanged(nameof(Current));
 this.OnPropertyChanged(nameof(CurrentCustomerIndex));
 ...
}
```

5. Open the `MainPage.xaml` file in the Design View window.
6. Add the `ProgressRing` control shown next to the top-level `Grid` control:

```
<Page
 ...
 Style="{StaticResource PageStyle}">

 <Grid>
 <ProgressRing HorizontalAlignment="Center" VerticalAlignment="Center"
 Foreground="AntiqueWhite" Height="100" Width="100"
 IsActive="{Binding IsBusy}" Canvas.ZIndex="1"/>
 <Grid x:Name="customersTabularView" Margin="10,20,10,20" Visibility="Visible">
 ...
 </Grid>
 </Grid>
</Page>
```

The `ProgressRing` control displays the rotating ring of activity that you may have seen when an application is busy. This control indicates to the user that there is activity—in other words, the application hasn't frozen. The control is only active when the `IsBusy` property of the `ViewModel` is `true`. Setting the `Canvas.ZIndex` property to 1 ensures that the progress ring appears in front of all other controls on the page when it's active.

7. Add the `TextBlock` control shown next to the end of the `customersTabularView` `Grid` control, after the control that displays the customer's phone number. This control will display the message held in the `LastError` property of the `ViewModel`.

```
...
<Grid x:Name="customersTabularView" Margin="40,54,0,0" Visibility="Visible">
 ...
 <Grid Grid.Row="2">
 ...
 <TextBox Grid.Row="5" Grid.Column="3" Grid.ColumnSpan="3" x:Name="phone" .../>
 <TextBlock Grid.Row="6" Grid.Column="1" Grid.ColumnSpan="7"
 Style="{StaticResource ErrorMessageStyle}" Text="{Binding LastError}"/>
 </Grid>
</Grid>
...
```

8. Find the `customersColumnarView` `Grid` control. Add a sixth `RowDefinition` control to the `Grid` control nested inside the `customersColumnarView` `Grid` control as follows:

```
...
<Grid x:Name="customersColumnarView" Margin="10,20,10,20" Visibility="Collapsed">
 ...
 <Grid Grid.Row="1">
 ...
 <Grid.RowDefinitions>
 <RowDefinition/>
 <RowDefinition/>
 <RowDefinition/>
 <RowDefinition/>
 <RowDefinition/>
 <RowDefinition/>
 <RowDefinition/>
 </Grid.RowDefinitions>
 </Grid>
 ...

```

9. At the end of the nested `Grid` control, add a `TextBlock` control that displays the message in the `LastError` property of the `ViewModel` class:

```
...
<TextBox Grid.Row="5" Grid.Column="1" x:Name="cPhone" .../>
<TextBlock Grid.Row="6" Grid.Column="0" Grid.ColumnSpan="2"
 Style="{StaticResource ErrorMessageStyle}" Text="{Binding LastError}"/>
</Grid>

```

**10.** In Solution Explorer, open the AppStyles.xaml file.

**11.** Add the following style to the end of the resource dictionary. The error message TextBlock controls use this style:

```
<Style x:Key="ErrorMessageStyle" TargetType="TextBlock">
 <Setter Property="HorizontalAlignment" Value="Left"/>
 <Setter Property="TextWrapping" Value="Wrap"/>
 <Setter Property="VerticalAlignment" Value="Top"/>
 <Setter Property="FontSize" Value="20"/>
 <Setter Property="Foreground" Value="White"/>
</Style>
```

**12.** Add the following Page.BottomAppBar control to the end of the page, immediately before the closing </Page> element. This control displays a command bar at the base of the page and indicates which record is currently displayed. The Run element of a TextBlock control enables you to display multiple text items as part of the same control.

```
<Page.BottomAppBar>
 <CommandBar>
 <AppBarElementContainer VerticalAlignment="Center">
 <TextBlock Padding="0, 0, 20, 0">
 <Run Text="{Binding Path=CurrentCustomerIndex}"/>
 <Run Text=" of "/>
 <Run Text="{Binding Path=CustomerListCount}"/>
 </TextBlock>
 </AppBarElementContainer>
 </CommandBar>
</Page.BottomAppBar>
```

You can now update the ViewModel class to call the web service to fetch customer data.

### To fetch data from the AdventureWorks web service

1. In Solution Explorer, right-click the **Customers** project and select **Manage NuGet Packages**.
2. In the NuGet Package Manager window, on the **Browse** tab, type **System.Text.Json**.
3. Select the **System.Text.Json** package (make sure it's version 6.0.0 or later) and select **Install**. Then, in the Preview Changes dialog, select OK.

The System.Text.Json package contains utilities for serializing and deserializing JSON text and converting it into objects in your application.

4. Open the ViewModel.cs file in the Code and Text Editor window.
5. Add the following using directives to the list at the top of the file:

```
using System.Net.Http;
using System.Text.Json;
```



6. In the `ViewModel` class, add the following private fields directly above the constructor. In the statement that initializes the `ServerUrl` variable, replace the string `https://yourwebservice.azurewebsites.net/` with the URL of your web service that you deployed in the previous exercise. (In my example case, the URL was `https://adventureworksservice20210729160556.azurewebsites.net/`.)

```
public class ViewModel: INotifyPropertyChanged
{
 ...
 public Command LastCustomer { get; private set; }

 private const string ServerUrl = "https://yourwebservice.azurewebsites.net/";
 private int offset = 0;
 private int count = 0;
 private HttpClient client = null;
 private JsonSerializerOptions options = new()
 {
 PropertyNameCaseInsensitive = true
 };

 public ViewModel()
 {
 ...
 }
 ...
}
```

7. Add the following code to the end of the constructor:

```
public ViewModel()
{
 ...
 this.client = new HttpClient();
 this.client.BaseAddress = new Uri(ServerUrl);
 this.client.DefaultRequestHeaders.Accept.Add(
 new MediaTypeWithQualityHeaderValue("application/json"));
}
```

These statements create and initialize a new `HttpClient` object. An application uses the `HttpClient` object type to send HTTP requests to a web service and obtain any response returned by the web service. You specify the address of the web service in the `BaseAddress` property, and indicate the type of data sent and received by setting the `DefaultRequestHeaders` property. In this application, you'll send and receive messages formatted as JSON strings.

8. Add the `GetDataAsync` method shown here to the `ViewModel` class, after the constructor:

```
public async Task GetDataAsync(int offset, int count)
{
}
```

This is an asynchronous method that will send a request to the GET operation in the web service that takes the offset and count parameters. The purpose of this method is to retrieve a block of customers and add them to the Customers list.

9. In the `GetDataAsync` method, implement the logic to retrieve customer information using the web service, as shown here in bold:

```
public async Task GetDataAsync(int offset, int count)
{
 this.offset = offset;
 this.count = count;
 try
 {
 this.IsBusy = true;
 var response = await this.client.GetAsync($"api/customers?offset={offset}&count={count}");
 if (response.IsSuccessStatusCode)
 {
 var customersJsonString = await response.Content.ReadAsStringAsync();
 var customersData =
 JsonSerializer.Deserialize<List<Customer>>(customersJsonString, options);
 if (this.customers is null)
 {
 this.customers = customersData;
 this.First();
 }
 else
 {
 this.customers.AddRange(customersData);
 }
 }
 else
 {
 this.LastError = response.ReasonPhrase;
 }
 }
 catch (Exception e)
 {
 this.LastError = e.Message;
 }
 finally
 {
 this.OnPropertyChanged(nameof(CustomerListCount));
 this.IsBusy = false;
 }
}
```

The key points in this code are as follows:

- The status of the `ViewModel` class is set to `Busy`. This action will cause the page to display the `ProgressRing` control.
- The `GetAsync` method of the client object sends an HTTP GET request to the web service. The request is sent to the `api/customers` endpoint. The `offset` and `count` parameters to the request are populated using the values provided to the `GetDataAsync` method.
- If the `GetAsync` method of the client object is successful, the data returned by the web service is deserialized from a JSON string into a collection of `Customer` objects.
- If the `customers` collection is currently `null`, then it is assigned this collection of `Customer` objects; otherwise, the data in this collection is appended to the `customers` collection.
- If an error occurs, the error message is assigned to the `LastError` property, which will be displayed on the page.
- In the `finally` block, the `Busy` status is reset (the `ProgressRing` control will disappear), and the record count is updated for display in the bottom app bar of the page.

10. Open the `MainPage.xaml.cs` file in the Code and Text Editor window.

11. Add the `BatchSize` constant shown next to the `MainPage` class. This value will be used to indicate how many rows should be retrieved at a time from the web service.

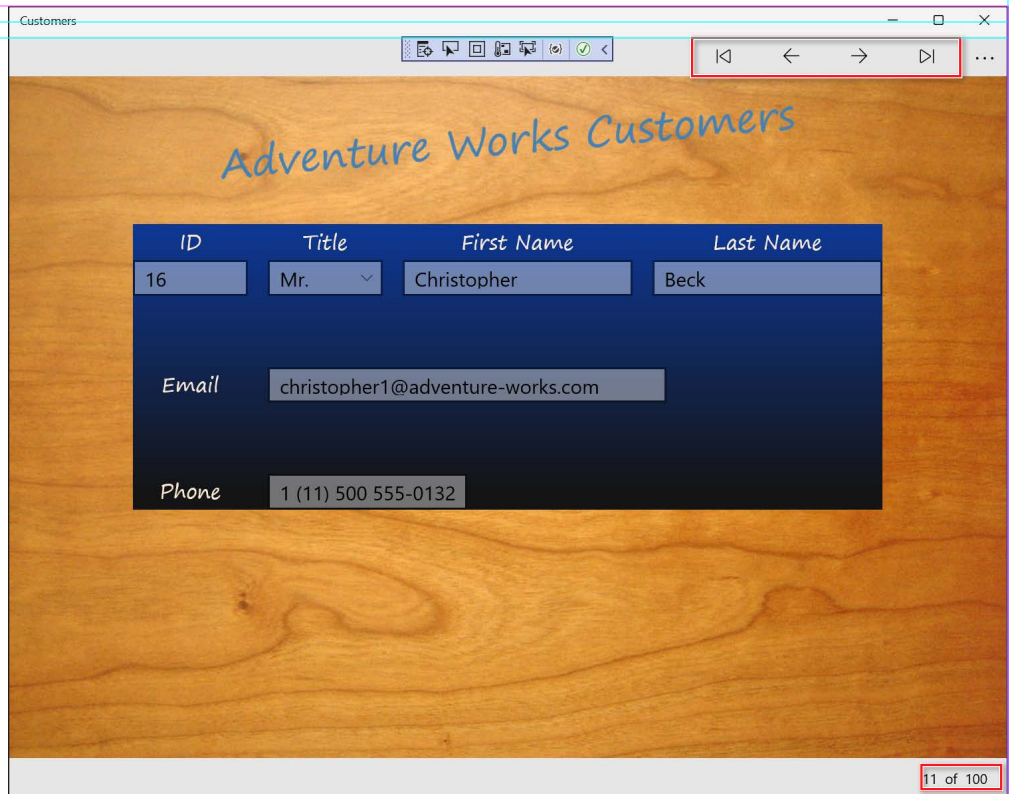
```
public sealed partial class MainPage : Page
{
 internal const int BatchSize = 100;
 ...
}
```

12. In the `MainPage` constructor, populate the `ViewModel` object by calling the `GetDataAsync` method. Set the `offset` parameter to 0 (to start at the beginning) and the `count` parameter to `BatchSize`:

```
public MainPage()
{
 ...
 ViewModel viewModel = new ViewModel();
 _ = viewModel.GetDataAsync(0, BatchSize);
 this.DataContext = viewModel;
}
```

13. On the **Debug** menu, select **Start Debugging** to build and run the application. When the application starts, you should momentarily see the `ProgressRing` control while the data is retrieved, and then the first customer should appear.

14. Select the buttons in the top app bar to move between customers. Notice that the count in the bottom app bar indicates which row you're positioned on.



15. When you've finished browsing the data, close the app and return to Visual Studio.

The Customers app fetches and displays the first 100 customers, but what if you want to see more? In the next exercise, you'll add this capability to the app. You can use the `GetDataAsync` method to do this and provide an appropriate `offset` to fetch the next block of customers. You also need to update the UI and add a More button.

### To display more customers and add a More button

1. Open the `ViewModel.cs` file in the Code and Text Editor window.
2. Add the `MoreCustomers` and `Reset` commands shown next in bold to the list of commands in the `ViewModel` class:

```
public class ViewModel : INotifyPropertyChanged
{
 ...
 public Command LastCustomer { get; private set; }
 public Command MoreCustomers { get; private set; }
 public Command Reset { get; private set; }
 ...
}
```

3. Add the `More` method shown next to the `ViewModel` class, after the `Last` method. This method fetches the next count records from the end of the current block records using the `GetDataAsync` method. The code updates the state of the `ViewModel` object to ensure that the navigation buttons continue to operate correctly:

```
private async Task More()
{
 await this.GetDataAsync(this.offset + this.count, this.count);
 this.currentCustomer = this.customers.Count > offset ? offset : this.customers.Count - 1;
 this.OnPropertyChanged(nameof(Current));
 this.IsAtStart = (this.currentCustomer == 0);
 this.IsAtEnd = (this.customers.Count == 0 || this.customers.Count - 1 == this.currentCustomer);
}
```

4. In the `ViewModel` constructor, initialize the `MoreCustomers` command to run the `More` method. Only enable the command if there's a valid `HttpClient` object available to retrieve the data.

```
public ViewModel()
{
 ...
 this.LastCustomer = new Command(this.Last, ...);
 this.MoreCustomers = new Command(async () => await this.More(),
 () => this.client is not null);
 ...
}
```

5. Similarly, initialize the `Reset` command to discard the existing data in the `Customers` collection and fetch the first block of customers again:

```
public ViewModel()
{
 ...
 this.MoreCustomers = new Command(...);
 this.Reset = new Command(async () => {
 this.customers = null;
 await this.GetDataAsync(0, MainPage.BatchSize);
 }, () => this.client is not null);
 ...
}
```

6. Open the `MainPage.xaml` file in the Design View window. Then add the following `AppBarSeparator` and `AppBarButton` controls to the end of the `TopAppBar` control. The user can click or tap the buttons to fetch the next block of customers or reset the page to its original state. The `AppBarSeparator` groups controls in the app bar together.

```
<Page.TopAppBar>
 <CommandBar>
 ...
 <AppBarButton x:Name="lastCustomer" .../>
 <AppBarSeparator/>
 <AppBarButton x:Name="more" Icon="More" Label="More"
 Command="{Binding Path=MoreCustomers}"/>
 <AppBarButton x:Name="reset" Icon="Rotate" Label="Reset"
```

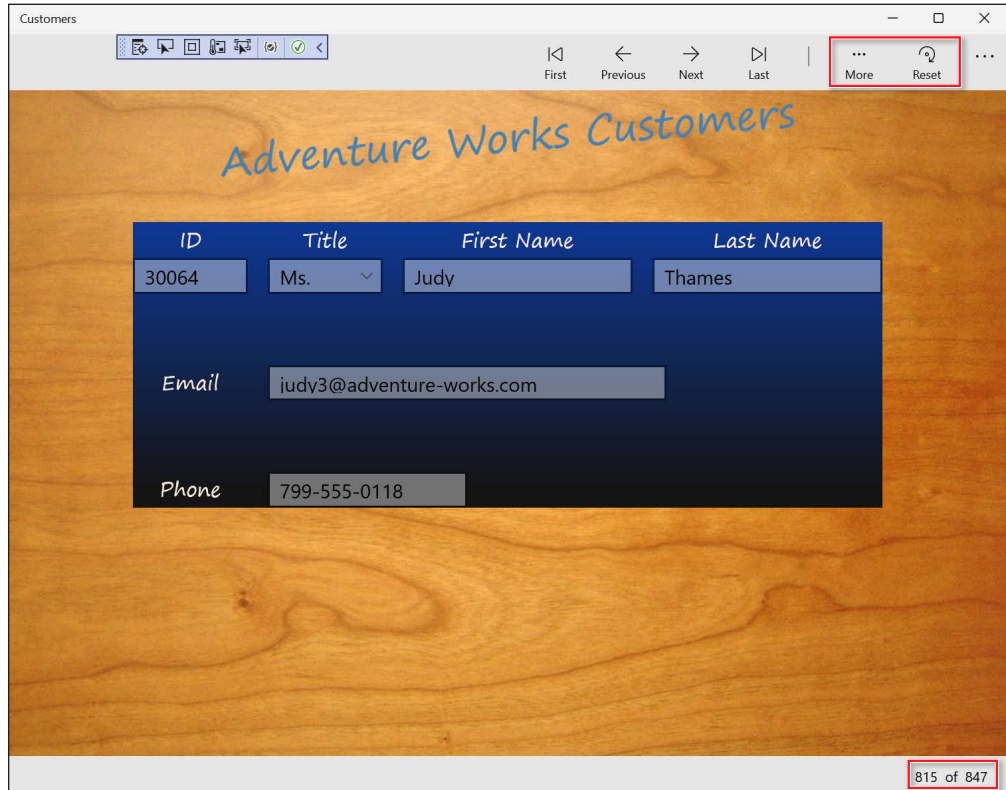
```

 Command="{Binding Path=Reset}"/>
 </CommandBar>
</Page.TopAppBar>

```

7. Build and run the application in debug mode.

The top app bar should now contain buttons for More and Reset. Each time you select More, an additional 100 rows are retrieved. If you select Reset, the existing list of customers is discarded, and the first 100 rows are retrieved again. Note that the count in the bottom app bar is updated to indicate how many rows have been retrieved each time you select More.



8. Return to Visual Studio and stop debugging.

## Searching for data in the Customers app

The Customers app enables you to browse through customer records, but sometimes you require a bit more refinement in the ability to locate a specific customer or set of customers. Many apps implement Query By Forms (QBF) features to address this requirement. Using QBF, you enter search criteria into the fields on the page and then select a Find button to find all matching rows. The AdventureWorks web service supports QBF through the `api/customers/find` endpoint that you created earlier. In this section, you'll examine how to implement QBF in the Customers app.

Initially, the app is used to browse data. I'll call this *browse* mode. However, QBF requires you to put the app into another mode, which, for the sake of argument, I'll refer to as *search* mode. In this mode, the page is cleared of data, and the user is expected to enter the values that indicate the search criteria. For example, the user might enter O% into the FirstName field on the page. When the user finishes entering the search criteria, the application can call the `api/customers/find` endpoint in the web service to retrieve a list of matching customers that the customer can then browse.

To support searching, you need to add properties that indicate whether the page is in browse mode or search mode. In browse mode, the user can navigate through the customers collection using the Next, Previous, First, Last, More, and Reset buttons. The user can also request to go into search mode. In search mode, the page is cleared. The user can then enter the data to use to find matching customers and select a Find button to actually find the data. At this point, the app returns to browse mode to allow the user to examine the data. In the following exercise, you'll implement browse and search mode and add buttons that enable the user to search for data.

### To implement browse and search modes in the ViewModel

1. In the Customers project, open the `ViewModel` class in the Code and Text Editor window.
2. Add the following Command objects shown in bold to the list at the top of the `ViewModel` class. These commands will enable the user to enter search mode, run a query, and return to browse mode, or just cancel search mode:

```
public int CurrentCustomerIndex { get => currentCustomer + 1; }
public Command SearchCustomers { get; private set; }
public Command RunQuery { get; private set; }
public Command CancelSearch { get; private set; }
public Command NextCustomer { get; private set; }
```

3. After the `More` method, add the enumeration and field shown next. The enumeration specifies the modes that the ViewModel supports, and the `EditMode` field will indicate the current mode:

```
private enum EditMode { Browsing, Searching };
private EditMode editMode;
```

4. After the `EditMode` field, add the following `IsBrowsing` and `IsSearching` properties. These properties get and set the mode of the ViewModel:

```
public bool IsBrowsing
{
 get => this.editMode == EditMode.Browsing;
 private set
 {
 if (value)
 {
 this.editMode = EditMode.Browsing;
 }
 this.OnPropertyChanged(nameof(IsBrowsing));
 this.OnPropertyChanged(nameof(IsSearching));
 }
}
```

```

public bool IsSearching
{
 get => this.editMode == EditMode.Searching;
 private set
 {
 if (value)
 {
 this.editMode = EditMode.Searching;
 }
 this.OnPropertyChanged(nameof(IsBrowsing));
 this.OnPropertyChanged(nameof(IsSearching));
 }
}

```

5. After the `IsSearching` property, add the `CanBrowse` and `CanSearch` properties shown next. These properties check the mode of the `ViewModel` and whether the `Customers` collection is valid:

```

private bool CanBrowse
{
 get => this.IsBrowsing &&
 this.client is not null;
}

private bool CanSearch
{
 get => this.IsSearching;
}

```



**Note** Strictly speaking, the `CanSearch` property is redundant, as it simply returns the value returned by the `IsSearching` property. However, it's good practice to keep properties in a `ViewModel` orthogonal, as it helps to make the intentions clearer when maintaining the code later.

6. In the `ViewModel` constructor, initialize the `SearchCustomers`, `RunQuery`, and `CancelSearch` commands. `Search`, `View`, and `Cancel` are methods that you'll create next. Note that the commands are only enabled when the `ViewModel` is in the appropriate mode.

```

public ViewModel()
{
 ...
 this.IsAtEnd = false;
 this.SearchCustomers = new Command(this.Search,
 () => this.CanBrowse);
 this.RunQuery = new Command(this.View,
 () => this.CanSearch);
 this.CancelSearch = new Command(this.Cancel,
 () => this.CanSearch);
 this.NextCustomer = new Command(this.Next,
 () => this.customers is not null && this.customers.Count > 1 && !this.IsAtEnd);
 ...
}

```



7. Update the `NextCustomer`, `PreviousCustomer`, `FirstCustomer`, `LastCustomer`, `MoreCustomers`, and `Reset` commands so that they are enabled only when the `ViewModel` is in browse mode:

```
this.NextCustomer = new Command(this.Next,
 () => this.CanBrowse &&
 this.customers is not null && this.customers.Count > 1 && !this.IsAtEnd);
this.PreviousCustomer = new Command(this.Previous,
 () => this.CanBrowse &&
 this.customers is not null && this.customers.Count > 0 && !this.IsAtStart);
this.FirstCustomer = new Command(this.First,
 () => this.CanBrowse &&
 this.customers is not null && this.customers.Count > 0 && !this.IsAtStart);
this.LastCustomer = new Command(this.Last,
 () => this.CanBrowse &&
 this.customers is not null && this.customers.Count > 0 && !this.IsAtEnd);
this.MoreCustomers = new Command(async () => await this.More(),
 () => this.CanBrowse && this.client is not null);
this.Reset = new Command(async () => {
 this.customers = null;
 await this.GetDataAsync(0, MainPage.BatchSize);
}, () => this.CanBrowse && this.client is not null);
```

8. Create the `FindCustomersAsync` method shown next. Add this method to the `ViewModel` class, after the constructor:

```
public async Task FindCustomersAsync(Customer pattern)
{
 try
 {
 this.IsBusy = true;
 var response = await this.client.GetAsync(
 $"api/customers/find?title={pattern.Title ?? "%"}" +
 $"&firstName={pattern.FirstName ?? "%"}&lastName={pattern.LastName ?? "%"}" +
 $"&email={pattern.EmailAddress ?? "%"}&phone={pattern.Phone ?? "%"}");

 if (response.IsSuccessStatusCode)
 {
 var customersJsonString = await response.Content.ReadAsStringAsync();
 customers = JsonSerializer.Deserialize<List<Customer>>(customersJsonString,
options);
 this.First();
 }
 else
 {
 this.LastError = response.ReasonPhrase;
 }
 }
 catch (Exception e)
 {
 this.LastError = e.Message;
 }
 finally
 {
 this.OnPropertyChanged(nameof(CustomerListCount));
 }
}
```

```

 this.IsBusy = false;
 }
}

```

This method is similar to the `GetDataAsync` method that you implemented previously. The main differences are as follows:

- This method takes a `Customer` object as its parameter. This object will contain the search criteria provided by the user.
- The code calls the `GetAsync` method of the `HttpClient` object to send a request to the `/api/customers/find` endpoint of the web service. The parameters passed to the request are retrieved from the `Customer` object. If any field is `null`, the wildcard character `%` is sent instead.

**9.** After the `FindCustomerAsync` method, define the `Search` method as follows:

```

private void Search()
{
 Customer searchPattern = new Customer { CustomerID = 0 };
 this.customers.Insert(currentCustomer, searchPattern);
 this.IsSearching = true;
 this.OnPropertyChanged(nameof(Current));
}

```

This method creates a new `Customer` object and makes it the current customer in the `Customers` list before putting the `ViewModel` into search mode. This action causes the customer to be displayed on the page. The user can then provide values and populate the `Customer` object using the page.

**10.** Create the `View` method shown next, after the `Search` method:

```

private void View()
{
 _ = FindCustomersAsync(Current);
 this.IsBrowsing = true;
 this.LastError = String.Empty;
}

```

This code calls the `FindCustomersAsync` method using the `Customer` object that the user has entered the search criteria in. The `ViewModel` is then placed back into browse mode to enable the user to view the results.

**11.** Add the `Cancel` method after the `View` method:

```

private void Cancel()
{
 this.customers.Remove(this.Current);
 this.OnPropertyChanged(nameof(Current));
 this.IsBrowsing = true;
 this.LastError = String.Empty;
}

```

The `Cancel` method removes the `Customer` object being used to enter search criteria from the `Customers` collection and puts the `ViewModel` back into browse mode.

## 12. Rebuild the solution. It should compile without any errors or warnings.

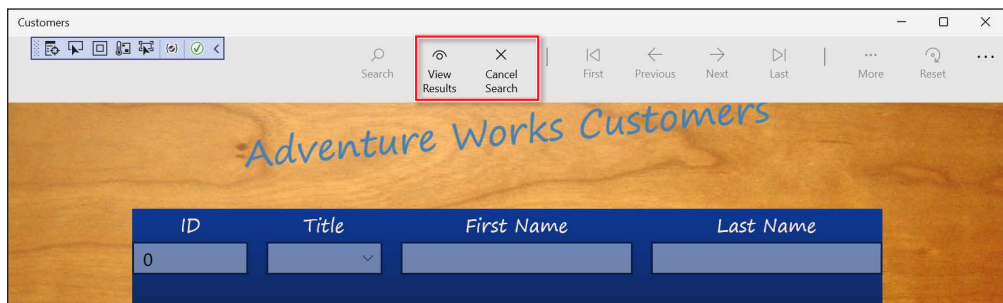
The final stage is to update the user interface and add buttons to the app bar that switch between search and browse modes.

### To update the user interface

1. Open the MainPage.xaml file in the design view window, and move to the Page.TopAppBar control near the end of the page.
2. Add the AppBarButton controls and AppBarSeparator shown next to the start of the CommandBar control. The AppBarButton controls are connected to the Command objects in the ViewModel object.

```
<Page.TopAppBar>
 <CommandBar>
 <AppBarButton x:Name="searchCustomers" Icon="Find"
 Label="Search" Command="{Binding Path=SearchCustomers}" />
 <AppBarButton x:Name="runQuery" Icon="View"
 Label="View Results" Command="{Binding Path=RunQuery}" />
 <AppBarButton x:Name="cancelSearch" Icon="Cancel"
 Label="Cancel Search" Command="{Binding Path=CancelSearch}" />
 <AppBarSeparator/>
 ...
 </CommandBar>
</Page.TopAppBar>
```

3. On the **Debug** menu, select **Start Debugging**.
4. When the app starts, in the top app bar, select the **Search** button (the one with a magnifying-glass icon). Notice that all buttons apart from View Results and Cancel are disabled.



5. In the **First Name** field, type **\_o%**. This pattern matches all customers that have the letter o as the second letter of their first name.
6. Tab to the **Last Name** field. Then, in the app bar, select the **Search** button. The query should return 136 rows. You can use the Next, Previous, First, and Last buttons to browse the matching customers.

Customers

Adventure Works Customers

ID	Title	First Name	Last Name
96	Ms.	Bonnie	Lepro

Email: bonnie2@adventure-works.com

Phone: 354-555-0130

11 of 136



**Note** It's important to tab away from the First Name field before selecting Search. If you don't leave the field, the page doesn't register the text as having been changed, and the query will perform a search for all records instead, returning 840 rows. There are ways around this feature, however. For example, you can capture each keystroke as it occurs in a field by using the `TextChanged` event. However, this technique is beyond the scope of this chapter.

7. Select **Search** again and enter search criteria into more than one field. When you select **View Results**, only records that match all the fields will be returned.
8. Select **Search** once more, and then select **Cancel**. The page should return to browse mode.
9. When you've finished, close the app and return to Visual Studio.

## Inserting, updating, and deleting data through a REST web service

In addition to giving users the ability to query and display data, many apps enable users to insert, update, and delete information. The ASP.NET web API implements a model that supports these operations through the use of HTTP PUT, POST, and DELETE requests. Conventionally, a PUT request

modifies an existing resource in a web service, and a POST request creates a new instance of a resource. A DELETE request removes a resource. The code generated by the Add New Scaffolded Item wizard in the ASP.NET web API template follows these conventions.

## Idempotency in REST web services

In a REST web service, PUT requests should be idempotent, which means that if you perform the same update repeatedly, the result should always be the same. In the case of the AdventureWorksService example, if you modify a customer record and set the telephone number to 888-888-8888, it doesn't matter how many times you perform this operation, because the effect will be identical.

This might seem obvious, but you should design a REST web service with this requirement in mind. With this design approach, a web service can be robust in the face of concurrent requests or even in the event of network failures. (If a client app loses the connection to the web service, it can simply attempt to reconnect and perform the same request again without being concerned about whether the previous request was successful.) Therefore, you should think of a REST web service as a means for storing and retrieving data, and you should not attempt to implement business-specific operations.

For example, if you were building a banking system, you might be tempted to provide a `CreditAccount` method that adds an amount to the balance in a customer's account and expose this method as a PUT operation. However, each time you invoke this operation, the result is an incremental credit to the account. Therefore, it becomes necessary to track whether calls to the operation are successful. Your app cannot invoke this operation repeatedly if it thinks an earlier call failed or timed out, because the result could be multiple duplicated credits to the same account.

For more information about managing data consistency in cloud applications, see "Data Consistency Primer" at <https://msdn.microsoft.com/library/dn589800.aspx>.

Many modern applications don't provide any functionality to delete certain types of data. For example, in the Customers app, you might want to ensure that you have a record of all customers that have done business with the Adventure Works organization, which might be required for auditing purposes. Additionally, even if a customer hasn't been active for a long time, there's a chance that the customer might place an order at some point in the future. In fact, it's becoming increasingly commonplace for business applications *never* to delete data, but simply to perform an update that marks the data as removed in some way and prevents it from being displayed. This is primarily because of the requirements to keep complete data records, often to meet regulatory requirements.

Sadly, time and space does not permit a detailed discussion describing how to implement insert and edit functionality in the Customers app. Suffice to say that you need to implement additional modes in the ViewModel (adding and editing) and create commands that invoke the relevant methods in the web service. However, a complete working version of the app that enables the user to create and update customers *is* available in the sample code supplied with this book. Look in the \Microsoft Press\VCBS\Chapter 27\Web Service\Customers with Insert and Update Features folder located in your Documents folder.

## Summary

In this chapter, you learned how to use the Entity Framework to create an entity model that you can use to connect to a SQL Server database. The database can be running locally or in the cloud. You also saw how to create a REST web service that a UWP app can use to query and update data in the database through the entity model. Finally, you learned how to integrate code that calls the web service into a ViewModel.

You have now completed all the exercises in this book. I hope you're thoroughly conversant with the C# language and understand how to use Visual Studio 2022 to build professional apps for Windows 11. However, this is not the end of the story. You've cleared the first hurdle, but the best C# programmers learn from continued experience—which you can gain only by building C# apps. As you do so, you'll discover new ways to use the C# language and many features in Visual Studio 2022 that I haven't had space to cover in this book.

Be aware that C# is an evolving language:

- In 2001, when I wrote the first edition of this book, C# introduced the syntax and semantics necessary to build apps that used Microsoft .NET Framework 1.0.
- Some enhancements were added to Visual Studio and .NET Framework 1.1 in 2003.
- In 2005, C# 2.0 emerged with support for generics and .NET Framework 2.0.
- C# 3.0 added numerous features, such as anonymous types, lambda expressions, and, most significantly, LINQ.
- C# 4.0 extended the language further, with support for named arguments, optional parameters, contravariant and covariant interfaces, and integration with dynamic languages.
- C# 5.0 added full support for asynchronous processing through the `async` keyword and the `await` operator.
- C# 6.0 provided further tweaks to the language, such as expression-bodied methods, string interpolation, the `nameof` operator, exception filters, and many others.
- C# 7 and C# 8 included many additional features, such as tuples, local functions in methods, support for expression-bodied members in properties and other scenarios, pattern matching in `switch` statements, the ability to handle and throw exceptions in new ways, enhanced syntax for defining numeric literals, and a general tidying up of the way that you can define and use out variables.
- The latest incarnations, C# 9 and C# 10, move the bar upward, adding many more syntactic features that you can use to make your code more concise and readable, as well as adding Record types.

In parallel with the evolution of the C# programming language, the Windows operating system has changed considerably since the first edition of this book. Arguably, the changes instigated by Windows 8 onward have been the most radical in this period. Developers familiar with earlier editions of Windows now have exciting new challenges to build apps for the modern, touch-centric, mobile platform that Windows 11 provides. Furthermore, modern business apps are extending beyond the

boundaries of the organization to the cloud, requiring you to implement highly scalable solutions that might need to support thousands or even millions of concurrent users. Visual Studio 2022, together with Azure and C#, will undoubtedly be instrumental in assisting you to address these challenges.

## Quick reference

To	Do this
Create an entity model by using the Entity Framework	Add a new class to your project and define properties for each column in the database table. Annotate the class with the <code>Table</code> attribute, and identify the column that acts as the primary key with the <code>Key</code> attribute. Create a <code>DbContext</code> class that the app can use to connect the model to a database.
Create a REST web service that provides remote access to a database through an entity model	Create an Azure API App using the ASP.NET Core Web application template. Run the Add Scaffold wizard and select API Controller With Actions, Using Entity Framework. Specify the name of the appropriate entity class from the entity model as the model class, and the data context class for the entity model as the data context class.
Deploy a REST web service to the cloud as an Azure API app	In Visual Studio, connect to your Azure subscription. Then use the Publish wizard to publish your web service as an Azure App Service. Specify an appropriate service plan that will support the volume of traffic that your web service expects to handle.
Consume a REST web service published as an Azure API app in a UWP application	Create an <code>HttpClient</code> object in the UWP application. Connect the <code>HttpClient</code> object to the URL of the web service. For example: <pre>const string ServerUrl =     "https://yourwebservice.azurewebsites.net/"; HttpClient client = null; client = new HttpClient(); client.BaseAddress = new Uri(ServerUrl);</pre>
Retrieve data from a REST web service in a UWP app	Call the appropriate GET method of the <code>HttpClient</code> object that's connected to the REST web service. For example: <pre>var serializedData = await client.GetAsync(\$"api/customers/5"); // The data is returned as a JSON string, which you can deserialize into an object: var customer = JsonSerializer.Deserialize&lt;Customer&gt;(serializedData);</pre>
Add a new data item to a REST web service from a UWP app	Use the appropriate POST method of the <code>HttpClient</code> object that's connected to the REST web service. Serialize the data as a JSON string, and pass this string, together with the appropriate URI in the REST web service, as parameters to this method. The value returned is a status code that indicates whether the operation was successful. For example: <pre>var serializedData = JsonSerializer.Serialize(myData); StringContent content = new     StringContent(serializedData, Encoding.UTF8, "text/json"); var response = await client.PostAsync("api/customers", content); if (response.IsSuccessStatusCode) {     // Post was successful } else {     // Handle the failure }</pre>
Update an existing item in a REST web service from a UWP app	Use the appropriate PUT method of the <code>HttpClient</code> object that's connected to the REST web service. Pass the key and the data for the modified data as parameters. For example: <pre>var serializedData = JsonSerializer.Serialize(myData); string path = \$"api/customers/{key}"; var response = await client.PutAsync(path, serializedData);</pre>

# Index

## SYMBOLS

`&=` compound assignment operator, 398  
`?` modifier  
    using with enumeration variables, 232  
    using with structure variables, 240  
`+=` operator, using with strings, 134  
`==` and `!=` operators, and structs, 547  
`+` (addition) operator, 60, 133  
`&` (AND) operator, 397–398  
`*` (asterisk) after file name, Visual Studio 2022, 19  
`@` (at sign), using with Circle objects, 206  
`\` (backslash), 129  
`{}` (curly braces)  
    matching, 17  
    terminology, 68  
`--` (decrement) operator, 69, 134, 542  
`/` (division) operator, 59  
`$` (dollar sign) symbol, using with strings, 60  
`"` (double quotation mark), 129  
`[]` ellipsis character, 41  
`=` (equal sign) assignment operator, 49, 112–113  
`==` (equal to) operator, 112–113  
`//` (forward slashes), using for comments, 24, 39  
`>` (greater than) operator, 113  
`>=` (greater than or equal to) operator, 113  
`++` (increment) unary operator, 69–70, 134  
`=>` (lambda) operator, 459  
`<` (less than) operator, 113  
`<=` (less than or equal to) operator, 113  
`<<` (left-shift) operator, 397–398  
`&&` (logical AND) operator, 113–114  
`*` (multiplication) operator, 59  
`~` (NOT) operator, 397

`!` (NOT) operator, 112, 114, 215  
`!=` (not equal to) operator, 112–113  
`|` (OR) operator, 397  
`||` (OR) operator, 113–114  
`()` (parentheses)  
    matching, 17  
    terminology, 68  
    wrapping expressions in, 524  
`+` (plus sign), meaning in book, xxix  
`;` (semicolon), using with `do` statements, 142  
`'` (single quotation mark), 129  
`-` (subtraction) operator, 59  
`[]` (square brackets)  
    terminology, 68  
    using with arrays, 252  
`~` (tilde), using with finalizers, 341  
`// TODO:` comments, 190, 208, 307–308, 617, 619–620  
`^` (XOR) operator, 398

## A

abstract classes. *See also* classes  
    implementing and using, 331–335  
    and methods, 329–330, 338  
abstract keyword, 337  
accessibility, controlling, 184–186  
Action type, using with tasks, 563  
Action<T, ...> and Func<T, ...> delegate types, 486  
adapter methods, 500  
AddCardToHand method, 465  
AddItemToLocalCache helper method,  
    Cache-Aside pattern, 613  
addition (+) operator, 60, 133



- addresses and customers, LINQ example, 514–516
- AddToAccumulator method, 581–582
- addValues method, 65, 74, 77–78
- ADO.NET, 728
- Adventure Works Customers app, 645–649, 718. *See also* UWP apps
- AdventureWorks database
  - Customer table, 723
  - installing, 719–724
  - removing unneeded columns, 724–728
  - setting up, 719–724
- AdventureWorks web API project, 729–730
- AdventureWorksService
  - fetching data from, 755–759
  - modifying customer records, 768
  - NuGet Package Manager, 731
  - operations, 736–744
- AggregateException class, task exceptions, 594–596
- aggregating data, 519–521
- Alt key. *See* keyboard shortcuts
- AND (&) operator, 397–398
- AND (&&) operator, 113–114
- anonymous classes, 202–203. *See also* classes
- App.config, Solution Explorer, 14
- application logic, developing, 85
- applications
  - building, 44
  - running in debug mode, 476–476
  - running without debugging, 19, 44
- AppStyles.xaml file, displaying, 676, 679, 681–682
- App.xaml source file, Solution Explorer pane, 38–39, 677
- ArgumentException class, 279
- argumentList in method call, 78
- arguments, passing to methods, 215. *See also* named arguments and parameters
- arithmetic operators
  - assignment operator, 68–69
  - associativity, 68–69
  - combining with assignment operators, 134
  - controlling precedence, 67–68
  - evaluating expressions, 68
  - examining, 61–67
  - implementing, 544–547
  - numeric types and infinite values, 61
  - remainder, 60
  - string interpolation, 60
  - and types, 59–61
- array arguments, using, 278–279
- array elements
  - accessing, 255–256, 275
  - iterating through, 275
- array instances, creating, 252–253
- array variables, declaring, 252, 275
- arrays. *See also* parameter arrays
  - accessing elements, 255–256
  - comparing to collections, 462–466
  - containing value types, 272–275
  - copying, 259–260
  - creating instances of, 275
  - finding number of elements in, 275
  - foreach statement, 256–257
  - implicitly typed, 254–255
  - vs. indexers, 395, 401–403
  - initializing elements of, 275
  - jagged, 261–271, 276
  - multidimensional, 260–271, 276
  - naming, 252
  - parameters in Main method, 258
  - parameters or return values for methods, 257–259
  - passing as parameters, 256–257
  - populating and using, 253–254
  - properties and indexers, 402
  - as reference types, 259
  - returning from methods, 258
  - System.Array class, 469
  - using square brackets ([]) with, 252
  - using to implement card game, 262–271
  - vs. variables, 251
- The Art of Computer Programming*, 420
- as operator, 225
- ASP.NET Core SSL Certificate, 741
- ASP.NET Core web API project, creating web services in, 735, 738
- ASP.NET web API, 740–741, 767–768

assignment operators. *See also* compound assignment operators  
 and associativity, 68–69  
 combining arithmetic operators with, 134  
 equal sign (=), 49

associativity  
 and assignment operator, 68–69  
 of operators, 538  
 using to evaluate expressions, 68

assemblies, and namespaces, 23

asterisk (\*) after file name, Visual Studio 2022, 19

async keyword, 603

async modifier, using with delegates, 606

asynchronicity and scalability, 600

asynchronous disposal, handing, 359–360.  
*See also* Dispose method; exception-safe disposal

asynchronous methods. *See also* PLINQ (Parallel LINQ); threads  
 ConfigureAwait(false), 610  
 defining and calling, 603  
 defining to return values, 609  
 efficiency, 613–614  
 IAsyncResult design pattern, 614  
 implementing, 639  
 memory allocation, 5  
 naming, 607  
 pitfalls, 610–611  
 problem, 600–603  
 solution, 603–608  
 tasks, 613–614  
 and Windows Runtime APIs, 611–613

asynchronous operations, overview, 599–600

at sign (@), using with Circle objects, 206

audit-*nnnnnn*.xml file, opening, 492

Auditor class, 506

AuditOrder method, 494

AuditService project, 493

automated factory scenario, 487–490

automatic properties. *See also* properties  
 creating, 394  
 defining, 386–387  
 generating, 381–383  
 and immutability, 385–387

await keyword, 603

await operator, asynchronous operations, 360, 587, 604–605, 609

Azure account, signing up for, 719

Azure API app, REST web service published as, 770

Azure SQL Database server  
 creating, 719–724  
 Firewall settings, 723

## B

background images, applying, 679

backslash (\), 129

BankAccount class, 374

Barrier class, 627

base classes, 314

base-class constructors, calling, 292–293, 310.  
*See also* constructors

BasicCollection<T> class, 478

binary notation, specifying integer values as, 410

binary operators, 538

binary tree class, building using generics, 423, 426–433

binary trees  
 building with generic methods, 434–436  
 greater than expression, 421  
 theory of, 420–423

binary values  
 displaying, 397  
 manipulating, 397–398  
 storing, 396

BinaryTree, retrieving data from, 525–529

binding properties of controls and objects, 716

BitArray class, 446

bits variable, using, 398–399

bitwise operators, 398

Black.Hole method, 282

Blank App template, Adventure Works  
 Customers app, 645–649

blocking wait operation, canceling, 640

blocks  
 using to group statements, 117–118  
 using with while statements, 136

- bold text, explained, xxix
- book
  - audience and starting point, xxvii–xxviii
  - conventions, xxviii–xxix
  - errata and support, xxxv
- bool data type, 50, 112
- bool keyword, type and structure, 237
- Boolean expressions
  - creating, 132
  - do statements, 142–143
  - while and for statements, 142
  - and while loops, 139
  - and while statements, 135
- Boolean operators
  - additive category, 115
  - assignment category, 115
  - conditional AND category, 115
  - conditional logical operators, 113
  - conditional OR category, 115
  - equality and relational operators, 112–113
  - equality category, 115
  - multiplicative category, 115
  - operator precedence and associativity, 114–115
  - pattern matching, 115
  - primary category, 114
  - relational category, 115
  - short-circuiting, 114
  - unary category, 114
- Boolean values
  - negating, 112, 114
  - true, 399
- Boolean variables
  - catchErrors, 158
  - declaring, 111–112, 132
- boxing and unboxing, 222–223
- boxing values, 230
- braces ({}), terminology, 68
- break
  - and continue statements, 143
  - and switch statements, 126
- breakpoints, inserting with Visual Studio Debugger, 94
- browse and search modes, ViewModel, 762–766

- Build Solution, selecting, 18
- built-in conversions, providing, 551
- Button class, 503
- Button control. *See also* More button
  - listing event names for, 40
  - separating business logic, 716
- Button element, 41
- buttons, adding to views in UWP apps, 713
- byte keyword, type and structure, 237

## C

- C#
  - case-sensitivity, 14, 46
  - as free format language, 46
  - history of, xxv–xxvi
- C# code, compiling, 18
- C# programs. *See* programs
- Cache-Aside pattern, 613–614
- cached and deferred evaluation, LINQ queries, 531–532
- calculateClick method, 79, 81, 163–164, 171
- CalculateFactorial function, 99
- CalculateInterest method, 381
- Calculator class, 352, 356
- camelCase notation, 48, 185
- CanBrowse property, 763
- Cancel method, adding, 765
- canceled tasks, using continuations with, 596.
  - See also* cooperative cancellation; tasks
- cancellation functionality, adding to
  - GraphDemo, 584–589
- cancellation of tasks, determining, 588–589, 592–594
- cancellation token, 582–584
- cancelling Parallel.For and Parallel.ForEach, 589
- CanExecute method, ViewModel, 706–708
- CanExecuteChanged method, ViewModel, 706
- CanSearch property, 763
- Card Game application, 263, 271
- card game
  - implementing using collections, 462–466
  - implementing with arrays, 262–271
- cascading if statements, 118–123

- case labels, using with switch statements, 125–126
- casting data safely, 224–226
- casting objects safely, 230
- catch block, 506
- catch handlers, using, 154–157
- catchall handler, using, 173
- catchErrors Boolean variable, 158
- catching exceptions, 50, 155, 157–158, 173–174, 177. *See also* try/catch statement block
- char data type, 50
- character pairs, matching, 17
- characters, reading from strings, 126–129
- checked expressions, writing, 166–168. *See also* expressions
- checked statements, writing, 166
- CheckoutButtonClicked method, 494, 498
- CheckoutController class, adding event to, 505–510
- CheckoutController component, creating, 495–499
- circle, calculating area of, 629–630
- Circle and Square classes
  - modifying, 334–335
  - using with interfaces, 322–328
- Circle class, 182–188, 198, 382–383
- Circle objects, using at sign (@) with, 206
- Class icon, IntelliSense, 17
- class keyword, using, 182–184
- class libraries, splitting into assemblies, 23
- class methods. *See* static methods and data
- class scope, defining, 83–84
- class types
  - declaring variables as, 206
  - handling, 205–206
- classes. *See also* abstract classes; anonymous classes; derived classes; interfaces; partial classes; sealed classes
  - and assemblies, 23
  - assigning, 293–295
  - and base classes, 314
  - copying, 205–211
  - creating hierarchy of, 300–305
  - defining, 204
  - defining and using, 182–184
  - and encapsulation, 182
  - fields in, 185
  - locating methods in, 56
  - vs. objects, 183
  - as reference types, 206
  - vs. structures, 238–240
  - using finalizer, 351–353
  - using namespaces with, 22
  - writing constructors for, 188
- classes and structures, comparing behaviors of, 246–247
- classification, understanding, 181–182
- CLI (command-line interface), 3
- Click attribute, 41
- Clone methods
  - using with arrays, 260
  - using with classes, 207–208
- cloud
  - deploying web services to, 746–751
  - importance of, xxvii
- CLR (common language runtime), 336, 600
- code
  - avoiding duplication in, 329
  - commenting out blocks of, 531
  - trying, 154–165
- Code and Text Editor window
  - opening, 54
  - using with graphical applications, 24–36
  - Visual Studio 2022, 13
- code samples, using, xxx–xxxv
- collection classes. *See also* concurrent collection classes; thread-safe collection
  - adding thread safety to methods in, 629
  - defining, 311–312
  - Dictionary<TKey, TValue>, 453–454
  - HashSet<T>, 455–457
  - LinkedList<T>, 449–450
  - List<T>, 447–449
  - making enumerable, 482
  - overview, 445–446
  - PriorityQueue<TElement, TPriority>, 451–452
  - SortedList<TKey, TValue>, 454–455
  - Stack<T>, 452–453
- collection initializers, using, 457–458

- collections. *See also* enumerating collections
  - adding items to, 467
  - comparing to arrays, 462–466
  - creating, 467
  - finding number of elements in, 467
  - iterating through elements of, 467
  - locating items in, 467
  - in memory, 522
  - removing items from, 467
  - using to implement card game, 462–466
- COM (Component Object Model), 100
- ComboBox control
  - and data binding, 699–701
  - data binding, 704–705
  - UWP app, 655–656
- command bars, resource for, 713
- Command class, implementing, 706–709
- command-line interface (CLI), 3
- commands
  - dir, 5, 7
  - dotnet run, 6
  - run, 6
- commenting code, 24. *See also* (forward slashes (//); // TODO: comments
- commenting out blocks of code, 531
- common language runtime (CLR), 336, 600
- compareClick method, 120
- CompareTo method, 312, 526
- compiling C# code, 18
- Complex struct, System.Numerics namespace, 544–547
- compound assignment operators, using, 133–134, 398, 541. *See also* assignment operators
- computer memory. *See also* memory allocation and efficiency
  - boxing, 222
  - organization of, 219–220
  - stack and heap, 220–221
  - System.Object class, 221
  - unboxing, 222–223
- concurrent collection classes. *See also* collection classes
  - adding thread safety, 629–638
  - and locks, 629–638
  - types, 628
- concurrent operations, being careful with, 622
- conditional logical operators, 113
- ConfigureAwait(false), asynchronous methods, 610
- Console Application template, Visual Studio 2022, 12–13
- console applications
  - creating using .NET CLI, 44
  - creating in Visual Studio 2022, 10–14
  - explained, 3
  - instance methods, 194–196
  - vs. UWP (Universal Windows Platform) app, 96
- Console class, 16, 277–278
- Console.Write method, 87
- Console.WriteLine method
  - data management, 521
  - parameter arrays, 277–278, 282
- Console.Write(v) statement, 94
- const field, declaring, 204
- const keyword, using, 199
- constructors. *See also* base-class constructors; deconstructor
  - calling, 201
  - declaring and calling, 204
  - deconstructing objects, 196–197
  - defaults, 186
  - defining, 384
  - overloading, 187–189
  - overview, 186–187
  - partial classes, 188
  - writing, 188–194
- continuations, using with tasks, 565, 596, 601–603
- continue and break statements, 143
- Continue button, Visual Studio Debugger, 93
- ContinueWith method, 564–566
- contravariant interfaces, defining, 441–444
- controller, adding to REST web service, 736–737
- controller classes, creating, 740–741
- controls
  - dragging on forms, 33
  - enabling for data binding, 716
  - enabling using data binding, 716

- conversion operators
  - defining, 550–555
  - writing, 553–555
- conversion operators, defining, 556. *See also* operators
- cooperative cancellation, 582–594. *See also* canceled tasks; tasks
- copying
  - arrays, 259–260
  - classes, 205–211
  - reference type variables, 229
  - reference types, 207–208
  - structure variables, 245–248
  - value type variables, 205–211, 229
- Count function, invoking, 524
- Count property, 712
- CountdownEvent class, 626
- covariant interfaces, defining, 440–441, 443–444
- CPU use, increasing for Task objects, 576
- CPU-bound code and Parallel class, 580
- .cs suffix, 5
- Ctrl key. *See* keyboard shortcuts
- curly braces ({}), matching, 17
- Current property, 704
- Customer class, INotifyPropertyChanged interface, 696–699
- Customer information
  - displaying, 689–694
  - modifying with two-way data binding, 694–699
  - ViewModel for, 702–705
- customers
  - displaying, 759–761
  - retrieving blocks of, 744–745
- customers and addresses, LINQ example, 514–516
- Customers app
  - extending, 768
  - scalable user interface, 649–659
  - searching for data in, 761–767
- Customers entity model, creating, 730–734
- Customers form
  - defining styles for, 675–685
  - Next and Previous buttons, 713–715

- CustomersController class, Scaffold wizard, 738–739

## D

- data
  - aggregating, 519–521
  - casting safely, 224–226
  - deleting through REST web services, 767–768
  - enumerating in order, 519, 534
  - filtering, 518–519, 523
  - grouping, 519–521, 523, 534
  - inserting through REST web services, 767–768
  - joining, 521–522, 535
  - locking, 625
  - ordering, 519–521, 523
  - retrieving, 519, 717–719
  - searching and sorting, 437
  - selecting for LINQ, 516–518
  - storage in LINQ, 515
  - synchronizing concurrent access to, 621–624
  - updating through REST web services, 767–768
- data access. *See* PLINQ (Parallel LINQ)
- data binding
  - ComboBox control, 704–705
  - controls and objects, 716
  - enabling controls for, 716
  - implementing for title ComboBox controls, 700–701
  - overview, 688
  - TextBox control, 704–705
  - using to modify data, 694–699
  - using with ComboBox control, 699–701
  - using to display Customer information, 689–694
  - UWP apps, 672
- “Data Consistency Primer,” 768
- data privacy and copying reference types, 207–208
- data types. *See* primitive data types
- databases
  - removing columns from, 724–728
  - retrieving data from, 717–719

- DataTypes project, 493
- Date struct, 243, 246
- dateCompare method, 120
- dates, comparing, 123
- DbContext class, Entity Framework, 732–733
- DbSets, defining, 733
- DealCardFromPack method, 464
- Debug menu. *See also* Visual Studio Debugger
  - arithmetic operators, 547
  - automatic properties, 387
  - calculating pi, 634–635
  - cancellation functionality, 587
  - Card Game application, 263, 271
  - card game using collections, 466
  - catching unhandled exceptions, 174
  - checked expressions, 167–168
  - CheckoutController class, 509
  - CheckoutController component, 499
  - class using finalizer, 352–353
  - Complex struct, 547
  - console applications, 18
  - constructors and objects, 192
  - conversion operators, 555
  - Customers app, 658–659
  - data binding, 693–695, 701
  - deferred and cached evaluation, 532–533
  - do statements, 150
  - enumerations, 235
  - exception handling, 169
  - expression-bodied methods, 76
  - fetching data from web service, 758
  - GraphDemo single-threaded application, 569, 575, 579, 607–608
  - graphical application, 41–42
  - hierarchy of classes, 303–305
  - IDisposable interface, 355
  - if statements, 119
  - indexers in Windows application, 405, 409–410
  - inheriting from DrawingShape class, 335
  - INotifyPropertyChanged interface, 696–699
  - InsertIntoTree method, 436
  - instance methods, 196
  - locks for serializing method calls, 638
  - MathOperators project, 61–62
  - method definitions, 76
  - Next and Previous buttons, 714
  - object initializers, 387
  - objects and disposal, 356
  - OperationCanceledException exception, 593
  - optional parameters, 106–108
  - parallelized LINQ query, 618, 620
  - params array and optional parameters, 287–288
  - preventing object disposal, 357
  - primitive data type values, 52–53, 58
  - propagating exceptions, 164
  - properties, 379
  - ref parameters, 218
  - retrieving data from BinaryTree, 527–530
  - Square and Circle classes, 328
  - static using statements, 202
  - structure types, 244
  - structures and classes, 247
  - styles for Customers form, 684
  - switch expressions, 131
  - tabular layout and Grid control, 666
  - task cancellation, 593
  - task status, 590
  - thread-safe collection, 637
  - throwing exceptions, 170–172
  - Tree<TItem> class, 432
  - try/catch statement block, 159
  - unhandled exceptions, 155, 159
  - user interface, 35
  - Utils.Sum method, 285
  - value and reference parameters, 209
  - verifying disposal of objects, 359
  - ViewModel for customer information, 705
  - Visual State Manager and layout, 674
  - while statements, 136
- debug mode, running applications in, 44
- Debug toolbar
  - commands on, 148
  - displaying, 91–92, 109
- debugging in Visual Studio 2022, 19, 41–42
- decimal data type, 50
- decimal keyword, type and structure, 237

- decision statements
  - Boolean operators, 112–115
  - Boolean variables, 111–112
  - if statements, 116–123
  - switch expressions with pattern matching, 129–131
  - switch statements, 124–129
- deconstructor, implementing, 196–197. *See also* constructors
- decrement (--) operators, 69, 134, 542
- decrementing and incrementing, 134
- decrementing variables, 69–70, 72
- default keyword, using with TreeEnumerator class, 472
- deferred and cached evaluation, LINQ queries, 531–532
- Delegate icon, IntelliSense, 17
- delegate type, declaring, 511
- delegates
  - automated factory scenario, 487–490
  - CheckoutController component, 495–499
  - creating instance of, 511
  - declaring and using, 490
  - Func<T, ...> and Action<T, ...>, 486
  - vs. function pointers, 485
  - invoking, 511
  - and lambda expressions, 499–500
  - in .NET class library, 485–486
  - purpose of, 483
  - understanding, 484–485
  - using async modifier with, 606
  - Wide World Importers application, 491–495
- Delegates project, 493–494
- DELETE button, UWP apps, 744
- DELETE request, 768
- DeleteCustomer method, 740
- deleting
  - data through REST web service, 767–768
  - unneded columns from database, 727
- DeliveryService project, 493
- Dequeue method, 416
- derived classes. *See also* classes
  - creating from base classes, 310
  - methods in, 295
- Design Patterns: Elements of Reusable Object-Oriented Software*, 500
- Design View window
  - displaying forms in, 41
  - Visual Studio 2022, 27–36, 672
  - zooming in and out of, 54
- destructors vs. finalizers, 342
- developer mode, enabling, xxx
- Developer Mode, using with graphical applications, 26
- Dictionary<TKey, TValue> collection class, 446, 453–454
- dir command, running, 5, 7
- Dispatcher object, 603–604
- dispatch-*nnnnn*.txt file, opening, 493
- disposal methods, calling, 346–347
- Dispose method. *See also* asynchronous disposal
  - calling from finalizer, 349–350
  - purpose of, 348
  - and thread safety, 357–358
- DistanceTo method, 194
- Distinct method, 524
- divideValues method, 66
- division (/) operator, 59
- DLLs (dynamic-link libraries), 5. *See also* SystemRuntime.dll assembly
- do statements, writing, 142–150
- doAdditionalProcessing method, 623, 633
- doAuditing method, 506
- Document Outline window, displaying, 63
- doFirstLongRunningOperation method, 605
- dollar sign (\$) symbol, using with strings, 60
- doMoreAdditionalProcessing method, 623
- doShipping method, 507
- dotnet run command, using, 6
- double data type, 50
- double keyword, type and structure, 237
- double quotation mark ("), 129
- double variable, initializing, 195
- doWork method, 244, 526
- DoWorkWithData method, 101
- Drawing Pad window, opening, 328, 380
- DrawingShape abstract class, 331–335



duplication in code, avoiding, 329  
dynamic-link libraries (DLLs), 5. *See also*  
    SystemRuntime.dll assembly

## **E**

ellipsis character [...], 41  
else statements, 116, 123  
encapsulation  
    implementing by using methods, 366–367  
    purpose of, 182  
Entity Framework  
    DbContext class, 732  
    mapping layer, 728  
    SQL SELECT commands, 728  
    using, 770  
    using with relational databases, 718  
    version availability, 731  
entity model, creating, 728–734, 770  
Enum icon, IntelliSense, 17  
Enumerable class, System.Linq namespace,  
    522–523  
enumerable collections  
    filtering rows from, 534  
    projecting fields from, 534  
enumerating collections, 469–470. *See also*  
    collections  
enumeration variables  
    assigning to values, 249  
    declaring, 249  
    nullable versions of, 232  
enumerations  
    choosing literal values, 233  
    choosing underlying types, 233–236  
    creating and using, 234–236  
    declaring, 231, 249  
    using, 232  
enumerators. *See also* IEnumerable interface  
    adding to Tree<TItem> class, 480–481  
    defining for Tree<TItem> class, 479–480  
    implementing, 470–474, 477–482  
    as pointers, 470  
    testing, 476–477, 481  
Equal method, 548  
equal sign (=) assignment operator, 49, 112  
equal to (==) operator, 112–113  
equality and relational operators, 112–113  
equality operators, overriding, 547–550  
equi-joins, support in LINQ, 524  
Error List window, displaying, 19, 548  
errors, occurrence of, 154  
errors and exceptions, overview, 153  
Event icon, IntelliSense, 17  
EventArgs argument, 504  
events  
    adding to CheckoutController class, 505–510  
    declaring, 501, 511  
    raising, 502–503, 512  
    subscribing to, 502, 511  
    unsubscribing from, 502, 512  
    user interface, 503–504  
    using, 504–510  
    using to enable notifications, 500–503  
Example class  
    defining class scope, 83  
    and IDisposable interface, 350  
    using enumeration, 232  
Exception family, 157  
exception handling, Visual Studio debugger,  
    169  
exceptions. *See also*  
    OperationCanceledException exception  
    catching, 154–155, 157–158, 177  
    filtering, 158–163  
    propagating, 163–164  
    raised by tasks, 598  
    throwing, 170–175, 177  
exception-safe disposal. *See also* asynchronous  
    disposal  
    class using finalizer, 351–353  
    finally block, 346–347  
    implementing IDisposable interface,  
        353–355, 362  
    preventing objects from disposed, 356–358  
    verifying object disposal, 358–359  
Execute method, ViewModel, 706–708  
Exists method, 485  
explicit conversion, 551–552  
expression-bodied methods, defining,  
    76–77, 109

expressions. *See also* checked expressions  
 catching, 172–173  
 comparing values of, 132  
 evaluating using associativity, 68–69  
 Extensible Application Markup Language (XAML). *See* XAML (Extensible Application Markup Language)  
 extension methods. *See also* methods  
 creating, 305–309  
 defining for types, 310  
 icon, 17  
 and LINQ, 517  
 using with BinaryTree, 525–529  
 Extract Method Wizard, 97. *See also* methods

## F

factorials, calculating, 98–100  
 factory scenario. *See* automated factory scenario  
 failing operations, retrying, 740  
 fall-through, stopping, 126  
 faulted tasks, using continuations with, 596.  
*See also* tasks  
 FFT (Fourier transform), 633  
 field names and properties, 369  
 fields. *See also* shared fields  
 hiding using methods, 366–367  
 projecting from enumerable collection, 534  
 projecting with Select method, 518  
 and variables, 83–84  
 file I/O, 612. *See also* I/O operations  
 FileOpenPicker class, 612  
 FileProcessor class, 341–342  
 files  
 creating, 5  
 opening, 137  
 filtering data, 523  
 finalizers  
 calling Dispose method from, 349–350  
 vs. destructors, 342  
 recommendations, 345  
 using with classes, 351–353  
 finalizers, writing, 340–342, 361  
 finally block  
 using, 175–177, 506  
 using with exception-safe disposal, 346–347  
 Find and Replace command, 55  
 Find method, 459, 485  
 FindCustomerAsync method, 764–765  
 FindCustomers method, 745  
 FinishFolding method, 500  
 Firewall settings, Azure SQL Database server, 723  
 first-in, first-out structure, 413–414  
 float data type, 50  
 float keyword, type and structure, 237  
 folders, creating, 719–714  
 for statements, writing, 140–142  
 foreach loop, using with grouped data, 520  
 foreach statement  
 calculating pi, 633  
 retrieving data from BinaryTree, 528  
 using with arrays, 256–257, 469  
 using with iterators, 479  
 using with List<T> collection, 447  
 form terminology, using with applications, 28  
 FormatException type, 155, 157, 160  
 forms. *See also* Windows Forms app  
 displaying in Design View, 41  
 dragging controls on, 33  
 Fortran, updates of, xxv  
 forward slashes (/), using for comments, 24, 39. *See also* commenting code  
 Fourier transform (FFT), 633  
 Frame object, creating, 39  
 free format language, C# as, 46  
 Func<T, ...> and Action<T, ...> delegate types, 486  
 function pointers vs. delegates, 485  
 functions vs. methods, 4

## G

Gamma, Erich, 500  
 garbage collection. *See also* objects  
 forcing, 361  
 function of, 344–345  
 using, 343–344, 351

- GC class, 357
- Generate Method Stub Wizard, using, 86–89, 109. *See also* methods
- generateGraphData method, 569, 573–574, 578–579, 585, 593
- generateGraphDataAsync method, 607–608
- generic classes
  - binary trees, 420–423
  - Tree<TItem>, 426–433
- generic interfaces and variance
  - contravariant interfaces, 441–443
  - covariant interfaces, 440–441
  - overview, 438–440
- generic methods. *See also* methods
  - creating, 433–434, 444
  - InsertIntoTree, 434–436
- generic types, using, 444
- generics
  - and constraints, 419
  - vs. generalized classes, 419
  - issues with object type, 413–416
  - solution, 417–418
  - using to build binary tree class, 423, 426–433
- gestures, UWP apps, 643
- get accessor, implementing, 406–407
- get and set keywords, using with properties, 368
- GetCachedeValue helper method, Cache-Aside pattern, 613
- GetCustomers method, 739–740
- GetDataAsync method, 756–757, 765
- GetEnumerator method, 470, 478
- GetHashCode method, overriding, 548
- globally unique identifier (GUID), 492
- Grade class, 385
- Grade struct, 388–389
- GraphDemo single-threaded application, 566–571
  - asynchronous method, 606–608
  - cancellation functionality, 584–589
  - modifying to use Task objects, 573–576
- graphical applications
  - adding code, 40–43
  - creating, 24–36
  - .NET MAUI (Multi-platform Application User Interface), 43
  - UWP (Universal Windows Platform) app, 37–39
  - Windows Forms app, 43
  - WinUI3, 42
  - WPF (Windows Presentation Foundation), 43
- graphical user interface (GUI), 3
- greater than (>) operator, 113
- greater than or equal to (>=) operator, 113
- Grid control
  - using to implement tabular layout, 659–667
  - using with Visual State Manager, 669–670
- Grid.Row attribute, adding to TextBlock control, 662
- GroupBy method, 520, 528
- grouping data, 519–521, 523
- GUI (graphical user interface), 3
- GUID (globally unique identifier), 492

## H

- hard disk space requirement, xxix
- HashSet<T> collection class, 455–457
- Haskell programming language, 458
- HeaderStyle, adding, 679
- heap and stack, using, 219–221
- “Hello World!” application
  - building, 3–6
  - using .NET CLI tools, 6–9
- Hello YourName! message, displaying, 14–21
- HelloUWP namespace, 38–39
- HelloWorld2.csproj file, 8
- Helm, Richard, 500
- helper methods, 97–100
- hexadecimal notation, specifying integer values as, 410
- hill-climbing algorithm, Windows Runtime (WinRT), 563
- Horse class, 329
- Hour struct, 539–540, 542–543, 552
- HTTP DELETE request, handling, 740
- HTTP POST request, responding to, 740
- HTTP PUT request, sending, 740
- HttpClient object, creating and initializing, 756

## I

- IAsyncOperation interface, 612
- IAsyncResult design pattern, 614–615
- IColor interface, defining, 320–322
- ICommand interface, ViewModel, 706
- IComparable<T> interface, 471–472, 475, 480, 526
- IDE (interactive development environment), 3
- idempotency in REST web services, 768
- identifiers, using, 45–46
- IDisposable interface
  - implementing, 353–355
  - and using statement, 347–350
- IDraw interface, defining, 320–322
- IEnumerable interface. *See also* enumerations
  - vs. IEnumerator, 470
  - implementing, 470, 475–477
  - and LINQ, 515
- IEnumerator<TItem> interface, 473
- if statements
  - accidental assignments, 117
  - nesting, 118–123
  - using, 116–123, 132, 136
  - writing, 119–123
- immutable properties, creating, 382
- immutable types, creating and instantiating, 394
- implicit conversion, 551
- implicitly typed local variables, declaring, 70–71. *See also* variables
- Important items, explained, xxviii
- increment operators, declaring, 542
- incrementing and decrementing, 134
- incrementing variables, 69–70, 72
- index accessors, 400–401
- indexers. *See also* interface indexer
  - vs. arrays, 395, 401–403
  - binary values, 396–398
  - calling, 407–409
  - creating for classes, 410
  - creating for structures, 410
  - defining in interfaces, 410
  - features of, 400
  - implementing, 411
  - in interfaces, 403–404
  - and operators, 538
  - overview, 395
  - properties and arrays, 402
  - solving problems, 398–400
  - using in Windows applications, 404–410
  - writing, 406–407
- inequality (!=) operator, 112–113
- Infinity value, 61
- inheritance
  - assigning classes, 293–295
  - calling base-class constructors, 292–293
  - calling base-class constructors, 310
  - declaring methods, 295–296
  - declaring override methods, 297–298, 310
  - declaring virtual methods, 296–297, 310
  - extension methods, 305–310
  - hierarchy of classes, 301–305
  - and interfaces, 330
  - overview, 289–290
  - protected access, 300–305
  - System.Object class, 292
  - using, 290–292
  - virtual methods and polymorphism, 298–299
- initializing variables, 211
- INotifyPropertyChanged interface, Customer class, 696–699
- InsertIntoTree method
  - testing, 436
  - writing, 434–436
- instance methods, writing and calling, 194–196
- int, referring to minimum value of, 165
- int data type, 50
- int keyword, type and structure, 237
- int types, 396
- int variable type, 49. *See also* long integers
- Int32 method, .NET libraries, 59
- IntBits struct, 399–400
- integer arithmetic, checked and unchecked, 165–168, 177
- integer values, displaying as binary or hexadecimal, 410
- Intel, 560
- IntelliSense, 15–17, 195
- interactive development environment (IDE), 3
- Interface icon, IntelliSense, 17

interface indexer, implementing, 411.

*See also* indexers

interface properties. *See also* properties

declaring, 375–381

implementing, 393

interfaces. *See also* classes

declaring, 338

declaring properties in, 393

defining, 312–313, 318, 320–328

explicitly implementing, 316–317

extending, 318

handling versioning with, 318–319

Hungarian notation, 313

IDraw and IColor, 320–322

implementing, 313–314, 338

indexers in, 403–404

and inheritance, 330

inheriting from, 314

overview, 311–312

reducing coding errors, 314

referencing classes through, 314–315

restrictions, 313, 319–320

Square and Circle classes, 322–328

working with, 315

Internet connection requirement, xxx

InvalidOperationException, 172

I/O operations, waiting for completion, 600.

*See also* file I/O

is operator

using, 224

using in pattern matching, 115

IsCardAlreadyDealt method, 464

IsSearching property, 763

IStoreWrapper<T> interface, 440

iterators, using to implement enumerators,  
477–481

IWrapper<T> interface, 439–440

## **J**

jagged arrays, creating, 261–271, 276

Johnson, Ralph, 500

join operator, using, 524

joining data, 521–522

## **K**

keyboard shortcuts

Quick Find dialog, 55

Visual Studio Debugger, 91–92

zooming in and out of Design View, 54

keywords, identifying, 46–47

Knuth, Donald E., 420

## **L**

lambda expressions

and anonymous methods, 461

body of, 459–460

and delegates, 499–500

forms of, 460–461

lambda (=>) operator, 459

(Language-Integrated Query (LINQ)), 599.

*See also* PLINQ (Parallel LINQ)

aggregating data, 519–521

commenting out blocks of code, 531

data storage, 515

and deferred evaluation, 530–533

equi-joins, 524

expressions in on clause, 524

filtering data, 518–519

grouping data, 519–521

joining data, 521–522

ordering data, 519–521

overview, 513–514

query operators, 522–524

querying data in Tree<TItem> objects,  
525–530

selecting data, 516–518

summarizing information, 524

using in C# application, 514–516

large methods, 97–100

layout, adapting using Visual State Manager,  
667–674

left-shift (<<) operator, 397–398

Length property, using with arrays, 256

less than (<) operator, 113

less than or equal to (<=) operator, 113

lightweight structures, implementing, 388–392

LinkedList<T> collection class, 446, 449–450

LINQ (Language-Integrated Query), 599.

*See also* PLINQ (Parallel LINQ)

aggregating data, 519–521

commenting out blocks of code, 531

data storage, 515

and deferred evaluation, 530–533

equi-joins, 524

expressions in on clause, 524

filtering data, 518–519

grouping data, 519–521

joining data, 521–522

ordering data, 519–521

overview, 513–514

query operators, 522–524

querying data in `Tree<TItem>` objects, 525–530

selecting data, 516–518

summarizing information, 524

using in C# application, 514–516

LINQ queries

forcing results for, 535

parallelizing, 639

parallelizing over collection, 616–618

parallelizing to join collections, 619–621

`List<T>` collection class, 446–449, 485

literal values, 49

local scope, defining, 83

local variables. *See* unassigned local variables

locking data, 625

locks

thread safety and `Dispose` method, 357–358

using to serialize method calls, 638

logical AND (&&) operator, 113–114

logical OR (||) operator, 113–114

long data type, 50

long integers, 50. *See also* `int` variable type

long keyword, type and structure, 237

loops, nesting, 135

## M

Main method

array parameters in, 258

and asynchronous methods, 604

calling `doWork` method, 189

using, 9

`MainPage` class, 509

`MainPage` constructor, 704

`MainPage` form, displaying and activating, 39

`MainPage.xaml.cs`, displaying, 37

`Mammal` class, 291–299

`ManualResetEventSlim` class, 626

`Math` class, 183, 197, 199

`MathOperators` application, performing calculations in, 63–66

`MathOperators` application, 61–62, 67

memory allocation and efficiency, 613–614.

*See also* computer memory

method calls, serializing with locks, 638

method definitions, viewing, 105

Method icon, IntelliSense, 17

method signature, explained, 295

`methodName`, 74, 78

methods. *See also* abstract classes; extension

methods; Extract Method Wizard;

Generate Method Stub Wizard; generic

methods; override methods; statements;

summary methods; virtual methods;

Visual Studio Debugger

accepting arguments, 288

call syntax, 77–80

calling, 109

constructors as, 186–187

declaring, 74–75, 109, 295–296

in derived classes, 295

and encapsulation, 366–367

explained, 73

expression-bodied methods, 76–77

finding, 458–460

vs. functions, 4

locating in classes, 56

nesting, 97–100

with optional parameters, 104–108

overloading, 84

overriding vs. hiding, 297

passing arguments to, 215

replacing with properties, 376–380

return types and void, 74

returning arrays from, 258

returning data from, 75

returning values from, 80–82, 109

methods (*continued*)

- specifying call syntax, 77–80
- specifying for tasks, 597
- stepping into and out of, 109
- syntax, 74
- tuples, 80–82
- and var keyword, 74
- variables in, 185
- writing, 85–89
- Microsoft IntelliSense, Visual Studio 2022, 15–17, 195
- Microsoft .NET and multitasking. *See also* tasks overview, 561
  - tasks, threads, and ThreadPool, 562–563
- Microsoft Press, contacting, xxxv
- Microsoft Windows platform, significance of, xxvi
- Min method, using with params array, 279–280
- models and views, communication between, 702
- Model-View-ViewModel (MVVM) pattern, implementing, 687–688
- Moore, Gordon E., 560
- Moore’s Law, 560–561
- More button, adding, 759–761. *See also* Button control
- More Info items, explained, xxviii
- MoveNext method, 473–474
- multicore processor, rise of, 560–561
- multidimensional arrays, using, 260–271
- multiplication (\*) operator, 59
- multiplyValues method, 66
- multitasking and parallel processing, 559–560
- multitasking using Microsoft .NET. *See also* tasks overview, 561
  - tasks, threads, and ThreadPool, 562–563
- MVVM (Model-View-ViewModel) pattern, implementing, 687–688
- MyFileUtil application, 259

## N

- named arguments and parameters, 100–108. *See also* arguments
- nameof operator, 699
- Namespace icon, IntelliSense, 17

- namespaces
  - and assemblies, 23
  - using, 21–22
- NaN (not a number) value, 61
- narrowing conversion, 551
- nested loops, creating, 135
- nested methods, creating, 109
- nesting
  - if statements, 118–123
  - methods, 97–100
- .NET class library
  - collection types, 446
  - exception classes, 170
- .NET CLI (command-line interface)
  - building and running projects, 6–9
  - creating console applications, 44
- .NET Framework, 11, 23, 625
- .NET libraries
  - and features, 336
  - Int32 method, 59
- .NET MAUI (Multi-platform Application User Interface), 43
- new keyword, 337
- Next and Previous buttons, ViewModel, 713–715
- NextCustomer command, ViewModel, 710–715
- NOT (~) operator, 397
- NOT (!) operator, 112, 114, 215
- not equal to (!=) operator, 112–113
- not a number (NaN) value, 61
- Notepad, accessing, 4
- Notes, explained, xxviii
- notifications, enabling by using events, 500–503
- NuGet Package Manager,
  - AdventureWorksService, 731
- null values and nullable types, 211–215, 229
- nullable types, properties of, 214–215
- null-coalescing operator, 213
- null-conditional operator, 212–214
- NullReferenceException exception, 489
- number generator, System. Random class, 252
- numeric types and infinite values, 61
- numeric values, specifying, 49–50. *See also* values

## O

- object initializers, using, 386–387
- object keyword, 221, 237
- object types and generics, 413–416, 438
- ObjectComparer object, 442
- objects. *See also* garbage collection
  - binding and data values, 716
  - binding properties of controls to, 716
  - casting safely, 230
  - vs. classes, 183
  - creating, 189–194, 201, 339
  - creating references to, 343
  - deconstructing, 196–197
  - enabling to notify bindings, 716
  - garbage collector, 343–345
  - initializing using properties, 383–387, 394
  - instantiating using generic types, 444
  - managing lifetimes of, 343
  - preventing from being disposed, 356–358
  - recommendations, 345
  - verifying disposal of, 358–359
  - writing finalizers, 340–342
- OK button, writing code for, 40–42
- onClick method, adding to MainPage class, 40–41
- OnPropertyChanged method, 697
- opening files, 137
- OperationCanceledException exception, 592–594. *See also* exceptions
- operator pairs, defining, 543–544
- operator precedence and associativity, summarizing, 114–115
- operators. *See also* conversion operators; symmetric operators
  - associativity, 538
  - comparing in structures and classes, 542–543
  - compound assignment evaluation, 541
  - constraints, 538
  - implementing, 544–547, 556
  - increment and decrement, 542
  - and indexers, 538
  - overloading, 539–540
  - precedence, 537
  - semantics, 537
  - symmetric, 540–541
  - understanding, 537–538
- optional parameters, using, 100–108
- optMethod method, 103
- OR (|) operator, 397
- OR (||) operator, 113–114
- OrderByDescending method, 519
- ordering data, 519–521, 523
- out and ref parameters, using, 215–218
- out parameters
  - creating, 216–218
  - passing arguments to, 229
- overflow checking, Visual Studio 2022, 165
- OverflowException, occurrence of, 169, 551
- overloaded operators, 539–540
- overloading
  - methods, 84
  - and parameter arrays, 277–278
- override keyword, 337
- override methods, declaring, 297–298, 310. *See also* methods

## P

- Parallel class
  - avoiding use of, 580–581
  - using to abstract tasks, 576–580
- Parallel LINQ (PLINQ). *See also* asynchronous methods; LINQ (Language-Integrated Query)
  - cancelling queries, 621
  - and declarative data access, 616
  - performance and collections, 616–621
- parallel processing and multitasking, 559–560
- parallel tasks. *See also* tasks
  - calculating pi, 634–636
  - loop iterations, 597
  - statement sequences, 597
- Parallel.For and Parallel.ForEach, canceling, 589
- parallelism
  - implementing using Task class, 566–576
  - increasing, 580
- parallelizing
  - declarative data access, 616–621
  - LINQ queries, 639



- ParallelPI method, 636
- ParallelTest method, 623–624
- parameter arrays. *See also* arrays
  - array arguments, 278–279
  - and Console.WriteLine method, 282
  - vs. optional parameters, 286–288
  - and overloading, 277–278
  - using, 277
- parameterList in method declaration, 74
- parameters
  - of containing types, 539
  - and named arguments, 100–109
  - naming parameters, 539
  - specifying by name, 102
- Parameters solution, opening, 208
- params array
  - declaring, 279–281
  - using, 283–285
  - writing, 283–284
- params object[], using, 281–282
- parentheses (())
  - matching, 17
  - terminology, 68
  - wrapping expressions in, 524
- partial classes, 188. *See also* classes
- Pass.Reference method, 211
- passwords, protecting, 734
- pattern matching
  - with properties, 380–381
  - using switch expressions with, 129–131
  - using, 115
- performance and PLINQ, 616–621
- Performance Profiler, GraphDemo single-threaded application, 571–573
- Person class, 272
- Person objects, array of, 272
- Person struct, 437, 458–459, 485
- Phone Book application, 405
- pi, calculating, 629–636
- PI field, Math class, 197
- PickMultipleFileAsync method, 612
- PickSingleFileAsync method, 612
- pixel (px) suffix, 32
- PlayingCard class, 264
- PLINQ (Parallel LINQ). *See also* asynchronous methods; LINQ (Language-Integrated Query)
  - cancelling queries, 621
  - and declarative data access, 616
  - performance and collections, 616–621
- PLINQ queries
  - canceling, 621
  - enabling cancellation in, 639
- plotButton\_Click method, 567, 574, 586, 606, 608
- plotXY method, 569
- plus sign (+), meaning in book, xxix
- Point class, 191–192, 196, 200
- pointers
  - enumerators as, 470
  - and memory management, 437–438
  - Span<T> type, 437–438
  - and unsafe code, 227–228
- pointsList collection, 636–637
- Polygon class, 386
- polymorphism and virtual methods, 298–299
- POST button, 744
- POST request, 768
- PostCustomer method, 740
- precedence of operators, 67–68, 72, 537–538
- predicates, finding, 458–460
- prefix and postfix, 70
- Previous and Next buttons, View Model, 713–715
- PreviousCustomer command, ViewModel, 710–715
- primitive data types
  - displaying values, 51–58
  - overview, 50, 205
  - unassigned local variables, 51
  - using in code, 53–58
- PriorityQueue<TElement, TPriority> collection class, 446, 451–452
- private and static keywords, writing, 201
- private fields, instance methods, 194
- private keyword, 337
- processor requirement, xxix
- Program class, 189
- Program.cs, Solution Explorer, 8–9, 14, 209–210

programming languages, updates of, xxv

programs

- using Visual Studio 2022, 14–21
- writing, 3–9

project file, example of, 5

projects

- adding, 495
- building and running with .NET CLI, 6–9
- locating items in, 55
- opening, 52

properties. *See also* automatic properties;  
 interface properties; read-only properties;  
 records with properties; write-only  
 properties

- accessibility, 372
- arrays and indexers, 402
- declaring, 368
- declaring in interfaces, 393
- expression-bodied members, 369
- and field names, 369
- getters and setters, 367–368
- implementing, 376
- modifying set accessors, 697
- overview, 367–370
- pattern-matching with, 380–381
- and public fields, 381
- read-only, 371
- read/write, 370, 393
- replacing methods with, 376–380
- restrictions, 373
- simulating assignment (=) operator, 538
- in Solution Explorer, 14
- and types, 378
- using, 370, 374, 376–380
- using to initialize objects, 383–387, 394
- write-only, 371–372, 393

properties of controls, binding to properties of  
 objects, 716

protected access, 300–305

protected keyword, 337

public and private keywords, using,  
 184–186, 300

public constructor, using, 186–187

public fields and properties, 381

public keyword, 337

PUT button, 744

PUT request, 768

PutCustomer method, 740

px (pixel) suffix, 32

Pythagoras' theorem, using, 195

## Q

QBF (Query By Forms), 761–762

query operators

- using, 522–524
- using with BinaryTree, 529–530

querying data in Tree<TItem> objects, 525–530

Queue class, 413–417

Queue<T> collection class, 446, 450–451

Quick Find dialog, displaying, 55

QuickWatch dialog, accessing, 161

## R

radius, declaring as private field, 184–185

RAM requirement, xxix

read access, sharing, 640

ReaderWriterLockSlim class, 626–627

ReadLine method, 87

read-only properties, 371. *See also* properties

read/write property, declaring, 370, 393

record types, defining and using, 390–392

records with properties, implementing

- lightweight structures, 388–392.

*See also* properties

ref and out parameters, using, 215–218

ref parameters

- creating, 216
- passing arguments to, 229
- using, 208–211, 217–218

reference type variables, copying, 229

reference types

- arrays as, 259
- classes as, 206
- copying, 207–208

reference variables, initializing, 211

References, Solution Explorer, 14

reflection and equality operators, 547

relational and equality operators, 112–113

- relational database
  - tables in, 522
  - and web services, 718
- remainder operator, 60
- remainderValues method, 66
- RenderTransform property, 681
- requestPayment method, 494
- resource management
  - disposal methods, 346
  - Dispose method and finalizer, 349–350
  - exception-safe disposal, 346–347
  - overview, 345
  - using statement and IDisposable interface, 347–348
- resources, releasing, 361
- response time, issues with, 599
- responsiveness, improving, 559
- REST web service. *See also* UWP apps; web services
  - accessing URLs, 735
  - adding controllers, 736–737
  - adding data items to, 770
  - creating and consuming, 770
  - creating and using, 735–744
  - data management, 767–768
  - deleting data, 767–768
  - and entity model, 770
  - HTTP protocol, 735
  - idempotency in, 768
  - inserting data, 767–768
  - publishing as Azure API app, 770
  - PUT requests, 768
  - remote access, 770
  - retrieving data from, 770
  - updating data, 767–768, 770
- result = clause in method call, 78
- RetrieveInOrder method, using with classes, 311
- RetrieveValue helper method, Cache-Aside pattern, 613
- return statements
  - adding to DistanceTo method, 195
  - using with methods, 74
  - and switch statements, 126
- returnType in method declaration, 74
- rows, filtering from enumerable collection, 534

- run command, using, 6
- Run To Cursor, using with do statement, 147
- running statements, 151

## S

- Save As command, using, 5
- sbyte keyword, type and structure, 237
- Scaffold wizard, CustomersController class, 738
- scalability, improving, 560
- scope
  - applying, 82
  - class scope, 83–84
  - local scope, 83
  - overloading methods, 84
  - and variables, 82
  - writing methods, 85–89
- ScreenPosition struct, 366–368, 371–372, 376
- sealed classes, creating, 330–335, 338.
  - See also* classes
- sealed keyword, 337
- search and browse modes, ViewModel, 762–766
- Search method, defining, 765
- searching
  - for data in Customers app, 761–767
  - enhancing in UWP apps, 745–746
  - and sorting data, 437
- Select method
  - and summary methods, 520–521
  - using with LINQ, 516–518
- semantics, defined, 46
- SemaphoreSlim class, 626
- semicolon (;), using with do statements, 142
- SerialPI method, 631–633, 636
- SerialTest method, 623–624
- server firewall, setting for AdventureWorks database, 722
- set accessors, using with properties, 379, 697
- set and get keywords, using with
  - properties, 368
- shared fields, creating, 198–199. *See also* fields
- short keyword, type and structure, 237
- short-circuiting, 114
- Show Steps, using with do statement, 147

- ShowAsync() method, 41
- showStepsClick method, 145
- single quotation mark ('), 129
- .sln suffix, 52
- slowMethod method, 601, 604
- Solution Explorer pane, Visual Studio 2022, 13–14
- Solution 'TestHello,' Solution Explorer, 13
- solutions, loading, 52
- SortedList<TKey, TValue> collection class, 446, 454–455
- sorting and searching data, 437
- Span<T> type, 437–438
- SQL (Structured Query Language), 514
- SQL SELECT commands, Entity Framework, 728
- square, calculating area of, 629–630
- Square and Circle classes
  - accessing members, 340
  - modifying, 334–335
  - testing, 326–328
  - using with interfaces, 322–326
- square brackets ([])
  - terminology, 68
  - using with arrays, 252
- stack and heap, using, 219–221
- Stack<T> collection class, 446, 452–453
- statement scope, understanding, 142
- statements. *See also* methods
  - associating with different values, 132
  - break, 143
  - continue, 143
  - do, 142–150
  - executing, 151
  - for, 140–142
  - grouping with blocks, 117–118
  - overview, 45–46
  - running, 151
  - while, 135–140
- static classes, understanding, 199
- static fields, declaring and accessing, 204
- static keyword, using to define methods, 208
- static members, writing, 200–202
- static methods and data
  - overview, 197–198
  - shared fields, 198
  - static classes, 199
  - static fields, 199
  - statis using statements, 200–202
- static methods, calling, 200–202, 204
- static properties, declaring, 371
- static using statements, 200–202
- Step Into, using with do statement, 148–150
- stepping through methods. *See* Visual Studio Debugger
- steps, showing, 147
- StopWatch type, 568
- string data type, 50
- string interpolation, 60, 72
- string keyword, type and class, 237
- string type, 206
- strings, reading characters of, 126–129
- Struct icon, IntelliSense, 17
- struct keyword, using, 237, 240
- structs, == and != operators, 547
- structure types
  - creating and using, 242–245
  - declaring, 249
- structure variables
  - copying, 245–247
  - declaring, 240, 249
  - initializing to values, 249
  - nullable versions of, 240
- Structured Query Language (SQL), 514
- structures
  - vs. classes, 238–240
  - declaring, 237–238
  - handling large ones, 248
  - initialization, 240–245
  - and operators, 238
  - types, 236–237
  - using, 238
- structures and classes, comparing behaviors of, 246–247
- styles, applying to UI, 674–685
- subtraction (–) operator, 59
- subtractValues method, 66
- summary functions, invoking, 524
- summary methods, using with Select method, 520–521. *See also* methods
- support and errata, xxxv
- Swagger generator, 733–734

- switch expressions, using with pattern matching, 129–131
- switch statements, using, 124–129, 225–226
- symmetric operators, creating, 540–541, 553.  
*See also* operators
- synchronization, canceling, 627–628
- synchronization primitives, coordinating tasks, 625–627
- synchronizing
  - access to shared pool of resources, 640
  - concurrent access to data, 621–624
  - tasks and implementing thread-safe access, 640
  - threads to wait for events, 640
- Synchronous I/O anti-pattern, 600
- syntax, defined, 46
- System namespace, 22
- system requirements, xxix–xxx
- System.Array class, 256, 469
- System.Collections.Generic namespace, 418, 442, 445
- System.Console class, 23
- System.Diagnostics.Stopwatch object, watch variable, 568
- System.IComparable and System.IComparable<T> interfaces, 424–425
- System.Linq namespace, Enumerable class, 522–523
- System.Numerics namespace, Complex struct, 544–547
- System.Object class, 221, 292
- System.Random class, 252
- System.Runtime.dll assembly, 23. *See also* DLLs (dynamic-link libraries)
- System.Threading namespace, 625

## T

- tabular layout, implementing using Grid control, 659–667
- TabularHeaderStyle style, 680
- Task class
  - features of, 562
  - using to implement parallelism, 566–576, 622

- task exceptions, handing using
  - AggregateException class, 594–596
- Task List window, using, 208
- Task objects
  - increasing CPU use, 576
  - using with GraphDemo application, 573–576
- Task type, using, 601
- Task<TResult> class, 609
- TaskContinuationOptions type, 565
- TaskCreationOptions enumeration, 564–565
- tasks. *See also* canceled tasks; cooperative cancellation; faulted tasks; multitasking using Microsoft .NET; parallel tasks
  - abstracting using Parallel class, 576–580
  - cancellation token, 582–584
  - confirming cancellation, 588–589
  - controlling, 563–565
  - cooperative cancellation, 582–594
  - creating, 563–565
  - creating and running, 597
  - creating continuations, 564
  - displaying status of, 590–592
  - enabling cancellation, 598
  - handling exceptions, 598
  - memory allocation and efficiency, 613–614
  - running, 563–565
  - scheduling, 564
  - specifying methods for, 597
  - synchronization primitives for coordination of, 625–627
  - synchronizing for thread-safe access, 640
  - threads and ThreadPool, 562–563
  - waiting for finishing, 597
- TemperatureMonitor class, 502
- TestHello, Solution Explorer, 14, 22
- text file viewer, creating, 136
- TextBlock controls
  - Grid.Row attribute, 662
  - vs. TextBox control, 30, 32, 34
  - using with Visual State Manager, 671
- TextBox controls
  - data binding, 704–705
  - using, 601
  - using with Visual State Manager, 671
  - UWP app, 651–657

## thread safety

- adding to methods in collection classes, 629
- and Dispose method, 357–358

threads. *See also* asynchronous methods

- and garbage collection, 344–345
- making wait for events, 640
- synchronizing, 640
- and ThreadPool, 562–563
- using to calculate pi, 631–634

thread-safe collection, 636. *See also*

- collection classes

## Thread.Sleep method, 582

## throw and switch statements, 126

## throw exceptions, using, 174–175

## tilde (~), using with finalizers, 341

## Time struct, 238–241

## Tips, explained, xxviii

## title ComboBox controls, implementing data

- binding for, 700–701

## // TODO: comments, 190, 208, 307–308, 617,

- 619–620. *See also* commenting code

## Toolbox, showing and hiding, 31

## ToString method, 243–244, 268, 296, 312, 390, 525

## Tree&lt;TItem&gt; class

- adding enumerator to, 480–481
- creating, 426–430
- IEnumerable<TItem> interface, 475–476
- testing, 430–433
- using iterator with, 479–481

## Tree&lt;TItem&gt; objects, querying data in, 525–530

## TreeEnumerator class, creating, 470–474

## triangle, modeling, 383–385

## Triangle public class, 383–384

## try block

- calculating pi, 632–633
- writing, 154–155, 592–593

## try/catch statement block, 159–163, 506.

- See also* catching exceptions

## tuples and methods, 80–82

## type inference and new operator, 184

## U

## UI (user interface)

- applying styles to, 674–685
- creating, 29–36
- error reporting, 752–755
- long-running operations, 752–755
- updating, 766–767
- updating for ViewModel, 766–767
- windows in, 28

## uint keyword, type and structure, 237

## Universal Windows Platform (UWP)

- device families, 642
- WinUI 3.0 and Win32 API, 642

## ulong keyword, type and structure, 237

## unary operators, 69, 538

## unassigned local variables, 51

## unhandled exceptions, 155–156, 159–163, 173–174

## user interface (UI)

- applying styles to, 674–685
- creating, 29–36
- error reporting, 752–755
- long-running operations, 752–755
- updating, 766–767
- updating for ViewModel, 766–767
- windows in, 28

## user-defined conversion operators, 552

## user-interface events, 503–504

## ushort keyword, type and structure, 237

## using directive

- explained, 22–23
- and IDisposable interface, 347–348
- static keyword in, 200–202
- thread safety and Dispose method, 358
- ViewModel, 708

## Util class, 279

## Utils.Sum method, testing, 285

## UWP (Universal Windows Platform)

- device families, 642
- WinUI 3.0 and Win32 API, 642

UWP apps. *See also* Adventure Works

- Customers app; REST web service; web services
- adapting layout, 667–674
- adding buttons to views, 713

UWP apps (*continued*)

- adding commands to ViewModel, 706–715
- adding More button, 759–761
- AdventureWorks REST web service
  - operations, 736–744
- AdventureWorks web API project, 729–730
- applying styles to UI, 674–685
- Azure SQL Database server, 719–724
- building with Blank App template, 645–649
- capabilities supported, 645
- connecting to, 719–723
- vs. Console apps, 96
- creating, 44, 686
- creating entity models, 728–734
- creating ViewModel, 702–706
- custom styles, 686
- Customers entity model, 730–734
- data binding, 672, 688–694, 699–701
- deploying web service to cloud, 746–751
- displaying customers, 759–761
- displaying data, 688–694
- enhanced searching, 745–746
- examining, 37–39
- features of, xxvi, 643–645
- form factors and orientations, 661–665
- gestures, 643
- implementing enhanced searching, 745–746
- lifetime, 643–644
- Model-View-ViewModel pattern, 687–688
- nameof operator, 699
- overview, 641–642
- Package.appxmanifest file, 644
- packaging, 644
- pages in, 28
- partial classes, 188
- removing unwanted columns, 724–728
- REST web service, 735–751
- restricted operations, 644
- retrieving block of customers, 744–745
- retrieving data from databases, 717–728
- running in debug mode, 36
- scalable user interface, 649–659, 686
- searching for data, 761–767
- structures created in, 388
- tabular layout with Grid control, 659–667

- testing, 666–667
- updating to use web services, 752–761
- uploading to Windows Store, 644
- user interface, 686
- Visual State Manager, 667–674
- Windows Runtime, 336

**V**

- value parameters, using, 208–201, 208–211
- value type variables, copying, 205–211, 229
- value types, explained, 205
- values. *See also* numeric values
  - assigning to variables, 132, 216
  - boxing and unboxing, 230
  - changing for variables, 133–134
  - comparing for expressions, 132
  - determining equivalency, 132
  - returning from methods, 80–82, 109
  - in variables, 48, 72
- ValueTask generic type, 614
- var keyword
  - and methods, 74
  - using with LINQ, 518
- variables. *See also* implicitly typed local variables
  - adding amounts to, 151
  - vs. arrays, 251
  - assigning values to, 72, 132, 216
  - camelCase notation, 48
  - changing values of, 72, 133–134
  - declaring, 48–49, 72, 206
  - and fields, 83–84
  - identifying as pointers, 227
  - incrementing and decrementing, 69–70, 72
  - initializing, 211
  - in methods, 185
  - naming, 48, 369
  - and scope, 82
  - specifying numeric values, 49–50
  - subtracting amounts from, 151
- variables defined with type parameter,
  - initializing, 475
- Variant type, 71
- Vehicles solution, opening, 301

- video card requirement, xxx
- View method, creating, 765
- ViewModel
  - adding commands to, 706–715
  - browse and search modes, 762–766
  - Command class, 706–709
  - control, 702–705
  - Next and Previous buttons, 713–715
  - NextCustomer command, 710–715
  - PreviousCustomer command, 710–715
  - XAML markup, 706
- views
  - adding buttons to, 713
  - and models, 702
- virtual keyword, 337, 375
- virtual methods. *See also* methods
  - declaring, 296–297, 310
  - and polymorphism, 298–299
- Visual State Manager, using to adapt layout, 667–674
- Visual Studio 2022
  - asterisk (\*) after file name, 19
  - building and running console application, 18–21
  - Code and Text Editor window, 13
  - configuring project dialog, 11–12
  - Console Application template, 12–13
  - console application, 10–14
  - console applications, 44
  - Design View window, 27
  - Error List window, 19
  - graphical applications, 25–28
  - Microsoft IntelliSense, 15–17
  - overflow checking, 165
  - overview, 9–14
  - Solution Explorer pane, 13–14, 20–21
  - Start Debugging command, 19
  - user interface, 29–36
  - UWP app, 44
  - writing programs, 14–21
- Visual Studio Community 2022 requirement, xxix
- Visual Studio Debugger. *See also* Debug menu; methods
  - amending code, 94–96
  - breakpoints, 94

- calculating factorials, 98–100
- Continue button, 93
- nesting methods, 97–100
- refactoring code, 96–97
- stepping through methods, 89–93
- Visual Studio debugger, exception handling
  - with, 169
- Vlissides, John, 500

## W

- Wait method, using with tasks, 594, 602
- wait operation, canceling, 640
- watch variable, System.Diagnostics.Stopwatch
  - object, 568
- web services. *See also* REST web service; UWP apps
  - building, 718
  - creating in ASP.NET Core web API project, 735
  - DELETE request, 768
  - deploying to cloud, 746–751
  - POST request, 768
  - and relational database, 718
  - updating UWP apps for, 752–761
  - updating UWP apps for, 752–761
- Where method, using with LINQ, 518–519
- where operator, using to filter data, 523
- while loop
  - creating, 138–139
  - rephrasing as for loop, 141
  - variable if in, 135
- while statements
  - structure, 140
  - writing, 135–140
- Wide World Importers application, 491–495
- widening conversion, 551
- Windows 10 requirement, xxix
- Windows application, using indexers in, 404–410
- Windows Forms app, 43. *See also* forms
- Windows Presentation Foundation (WPF)
  - graphical app, 43
- Windows Runtime APIs, asynchronous
  - methods, 611–613



- Windows Runtime (WinRT)
  - compatibility with, 336–337
  - hill-climbing algorithm, 563
  - iterative strategy, 563
  - overview, 641
  - tasks and schedules, 562
- WinUI3 graphical application, 42
- WPF (Windows Presentation Foundation)
  - graphical app, 43
- WrappedInt object, 211
- Wrapper struct, 402–403
- write access, making exclusive, 640
- WriteLine function, 4
- WriteLine method, 16–17
- write-only properties, 371–372, 378, 383, 393.
  - See also* properties
- writing
  - and calling instance methods, 194–196
  - checked expressions, 166–168
  - checked statements, 166
  - constructors, 188–194
  - conversion operators, 553–555
  - and creating objects, 189–194
  - do statements, 142–150
  - finalizers, 340–342, 361

- if statements, 119–123
- indexers, 406–407
- InsertIntoTree method, 434–436
- methods, 85–89
- params array method, 283–284
- private and static keywords, 201
- for statements, 140–142
- static members, 200–202
- try block, 592–593
- try/catch statement block, 159–163
- while statements, 135–140

## X

- XAML (Extensible Application Markup Language), 28–29, 31, 33, 96
- XAML markup
  - Next and Previous buttons, 714
  - UWP app, 647–648, 650, 652, 654–657, 662–665, 668–673, 675–676, 678–681, 685
  - ViewModel, 706
- XML representation, mapping characters to, 126–129
- XOR (^) operator, 398